

SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH

FAST MESSY GENETIC ALGORITHMS

**(SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED
PEPTIDES BY DETECTING SECONDARY STRUCTURES)**

THESIS

Steven R. Michaud, Captain, USAF

AFIT/GCS/ENG/01M-06

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

20010706 150

AFIT/GCS/ENG/01M-06

SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH FAST
MESSY GENETIC ALGORITHMS
(SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED
PEPTIDES BY DETECTING SECONDARY STRUCTURES)

THESIS

Steven R. Michaud
Captain, USAF

AFIT/GCS/ENG/01M-06

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, of the Department of Defense, or the United States Government.

AFIT/GCS/ENG/01M-06

SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH
FAST MESSY GENETIC ALGORITHMS
(SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED
PEPTIDES BY DETECTING SECONDARY STRUCTURES)

THESIS

Presented to the Faculty of the Graduate School of Engineering and Management,

Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Computer Science

Steven R. Michaud, B.S., M.S.

Captain, USAF

March, 2001

Approved for public release; distribution unlimited

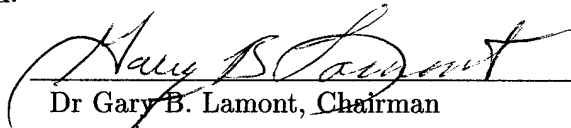

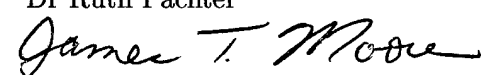
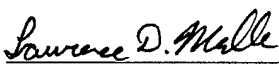
SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH
FAST MESSY GENETIC ALGORITHMS

(SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED
PEPTIDES BY DETECTING SECONDARY STRUCTURES)

Steven R. Michaud, B.S., M.S.

Captain, USAF

Approved:

 Dr Gary B. Lamont, Chairman	<u>25 FEB 01</u>
 Dr Ruth Pachter	<u>27 Feb 01</u>
 Dr James T. Moore	<u>28 Feb 01</u>
 Dr (Maj) Laurence D. Merkle	<u>28 Feb 01</u>

Acknowledgements

I would like to start out by thanking my wife, [redacted] for her continued support—taking care of Kendrick the Terror (our son) for hours on end giving me the freedom to undertake this thesis effort. I would like to thank my son for helping me forget about the trials and tribulation of the day and instead see life from his perspective—be it playing his Star Wars©, Tonka©, or NASCAR©toys.

I have my wife and son to thank at home, but there are a number of individuals that I worked with on a daily basis who played a very significant role in aiding me during this herculian task. To my fellow students, David Strong, Barry Secrest, and Jesse Zydallis, I know I couldn't have done this without your help. David, you have my sincerest thanks for the numerous hours you spent over my shoulders while trying to convert the software from the Paragon NX parallel platform to that of MPI. Barry, I enjoyed our discussions on the many different ways to attack/solve problems, the help with homework, and the occasional diversionary chess game, and finally Jesse, to whom I owe a tremendous debt for helping me publish a number of papers and for helping me to understand the fast messy Genetic Algorithm—thank God you had to learn it before I did.

Finally, I would like to thank Dr. Gary Lamont, for his understanding of my research, his continued support and his willingness to provide continual feedback and not minding too much if I was late in turning in any homework/project assignments. I am also indebted to my sponsor, Dr. Ruth Pachter for taking the time to help me better understand microbiology and the basics of chemistry. I would also like to acknowledge the other members of my committee, Dr. James Moore and Major Larry Merkle both of whom provided constructive feedback, making this document that much better.

Steven Ronald Michaud 06 March 2001

Steven R. Michaud

Table of Contents

	Page
Acknowledgements	iii
List of Figures	xii
List of Tables	xvii
Abstract	xx
 I. Introduction	 1-1
1.1 Protein Structure Prediction: A Grand Challenge Problem .	1-1
1.1.1 PSP Problem Class	1-2
1.1.2 PSP Structure	1-3
1.2 (Search and Parallel) Algorithms	1-5
1.2.1 Search Algorithms	1-5
1.2.2 Parallel Algorithms	1-6
1.3 Objectives	1-7
1.4 Approach	1-8
1.5 Assumptions	1-8
1.6 Scope	1-9
1.7 Layout of the Thesis	1-9
1.8 Conclusion	1-10
 II. Background: Literary Review and Requirements	 2-1
2.1 Introduction	2-1
2.2 The Protein Structure Problem	2-1
2.2.1 Introduction to Proteins and Associated Terminology	2-2
2.2.2 Experimental Tertiary Structure Determination . . .	2-4
2.2.3 Tertiary Structure Prediction (TSP)	2-4

	Page
2.2.4 Classical Prediction Methods	2-5
2.2.5 Other Prediction Methods	2-6
2.2.6 Met-Enkephalin and Polyalanine	2-7
2.2.7 PSP Landscape	2-9
2.2.8 PSP Energy Models	2-13
2.2.9 Bounding the Search Space	2-17
2.3 Landscape Visualization	2-26
2.4 Deterministic Search Algorithms	2-27
2.5 Stochastic Search Algorithms	2-28
2.5.1 Genetic Algorithms (GAs)	2-28
2.5.2 The Schema Theorem and Deception	2-30
2.5.3 Messy Genetic Algorithm (mGA)	2-31
2.5.4 Fast Messy Genetic Algorithm (fmGA)	2-35
2.6 Parallel Architectures and Genetic Algorithms	2-36
2.6.1 Homogeneous vs. Heterogeneous Computing Environ- ments	2-36
2.6.2 Parallel Genetic Algorithms	2-37
2.6.3 Parallel Decomposition Techniques	2-37
2.6.4 Island and Neighborhood Model	2-38
2.6.5 Typical Metrics to Compare Parallel Systems	2-39
2.7 Random Number Generators (RNG)	2-40
2.8 Summary	2-40
III. Methodology: Incorporation Secondary Structure into the fmGA . . .	3-1
3.1 Introduction	3-1
3.2 Background	3-1
3.2.1 AFIT's PSP fmGA Implementation	3-1
3.2.2 Evolution of AFIT's GAs	3-2

	Page
3.2.3 Inputs into AFIT's PSP fmGA	3-3
3.3 Modifications-Additions To AFIT's PSP fmGA	3-9
3.3.1 Porting the fmGA from NX to MPI	3-9
3.3.2 Fine Tuning the Local Search	3-10
3.3.3 Heterogeneous Version of fmGA	3-11
3.3.4 The Epoch fmGA	3-13
3.3.5 Searching for Secondary Structures	3-13
3.3.6 Incorporating Ramachandran Constraints into AFIT's PSP fmGA	3-14
3.3.7 Short-circuiting the Energy Fitness Function	3-18
3.3.8 Decimal Representation of Dihedral Angles as Output	3-21
3.3.9 Scaling the fmGA to larger Proteins	3-22
3.4 Summary	3-22
IV. Experimental Design	4-1
4.1 Introduction	4-1
4.2 General	4-3
4.2.1 Test Proteins	4-3
4.2.2 General Data Requirements	4-4
4.2.3 Random Number Seed	4-4
4.2.4 Root Mean Squared Deviation (RMSD)	4-6
4.2.5 Test Platforms	4-6
4.2.6 Experiment Specifics	4-7
4.3 Statistical Testing and Techniques	4-9
4.4 Experiments	4-9
4.4.1 Experiment 1: Parallel vs Serial fmGA	4-10
4.4.2 Experiment 2: Generating Competitive Templates with the Hill-Climbing Sweep Function	4-10

	Page
4.4.3 Experiment 3: Random Search with and without the Hill-Climbing Sweep Function	4-11
4.4.4 Experiment 4: Improving Performance in a Heterogeneous Environment	4-11
4.4.5 Experiment 5: Searching for a Secondary Structure .	4-13
4.4.6 Experiment 6: Effects of Seeding the Population . .	4-14
4.4.7 Experiment 7: Incorporating the Epoch Idea	4-15
4.4.8 Experiment 8: Short-circuiting the Energy Fitness Function	4-15
4.4.9 Experiment 9: Effects of Ramachandran Constraints	4-15
4.5 Summary	4-16
V. Results and Analysis	5-1
5.1 Introduction	5-1
5.2 Experiment 1: Parallel vs Serial fmGA	5-1
5.2.1 Results and Analysis: Serial	5-1
5.3 Experiment 2: Generating Competitive Templates with a Hill-Climbing Sweep Function	5-15
5.3.1 Results	5-15
5.3.2 Analysis	5-17
5.4 Experiment 3: Random Search with and without the Hill-Climbing Sweep Function	5-17
5.4.1 Results	5-17
5.4.2 Analysis	5-19
5.5 Experiment 4: Improving Performance in a Heterogeneous Environment	5-20
5.5.1 Results	5-20
5.5.2 Analysis	5-23
5.6 Experiment 5: Searching for a Secondary Structure	5-23

	Page
5.6.1 Results	5-23
5.6.2 Analysis	5-23
5.7 Experiment 6: Effects of Seeding the Population	5-25
5.7.1 Results	5-25
5.7.2 Analysis	5-26
5.8 Experiment 7: Incorporating the Epoch Idea	5-29
5.8.1 Results	5-29
5.8.2 Analysis	5-29
5.9 Experiment 8: Short-circuiting the Energy Fitness Function .	5-31
5.10 Experiment 9: Effects of Ramachandran Constraints	5-32
5.10.1 Results	5-32
5.10.2 Analysis	5-32
5.11 Final Observations	5-33
5.11.1 RMSD	5-33
5.11.2 Kruskal Wallis Test	5-36
5.12 Summary	5-40
VI. Conclusions	6-1
6.1 Observations	6-1
6.2 Future Considerations	6-2
Appendix A. Porting from NX to Message Passing Interface (MPI)	A-1
A.1 Introduction	A-1
A.2 NX communications system calls	A-1
A.3 MPI communication routines	A-1
Appendix B. Parallel Platforms and Computing	B-1
B.1 Introduction	B-1
B.2 Parallel Computing	B-6

	Page
B.2.1 Parallel Architectures	B-6
B.2.2 Parallel Algorithms	B-8
Appendix C. IBM's "Blue Gene" Supercomputer	C-1
C.1 Computer	C-1
Appendix D. Protein Structure Prediction Details	D-1
D.1 Background on the Protein Folding and Protein Structure Prediction Problems	D-1
D.2 Introduction to Proteins and Associated Terminology	D-1
D.2.1 Chemical Underpinnings of Proteins	D-1
D.2.2 Levels of Structure in the Protein Architecture	D-3
D.2.3 Forces at Play	D-7
D.2.4 Conformation of a Protein	D-7
D.3 Experimental Tertiary Structure Determination	D-9
D.3.1 X-ray crystallography	D-9
D.3.2 Nuclear Magnetic Resonance Spectroscopy	D-10
D.4 Tertiary Structure Prediction (PFP)	D-11
D.5 Classical Prediction Methods	D-11
D.6 Other Prediction Methods	D-15
D.7 Summary	D-15
Appendix E. Load Balancing pfmGA PSP Software	E-1
E.1 Introduction	E-1
E.2 Improving Processor Utilization	E-1
E.2.1 Asynchronous Load Balancing	E-2
E.2.2 Systems Performance Analysis and Load Balancing	E-2
E.2.3 Dynamic Comparison Performance Load Balancing	E-2
E.2.4 System Performance Analysis/Dynamic Comparison Load Balancing	E-4

	Page
E.3 Conclusions	E-5
Appendix F. Determining Population Size	F-1
F.1 Introduction	F-1
Appendix G. Evolutionary Computation	G-1
G.1 Introduction	G-1
G.1.1 Brief History of Evolutionary Algorithms	G-1
G.2 Genetic Algorithms	G-2
G.2.1 Origins of Genetic Algorithms	G-4
G.3 Simple Genetic Algorithm (SGA)	G-4
G.3.1 Simple Genetic Algorithm Operators	G-6
G.3.2 Simple Genetic Algorithm Parameters	G-7
G.3.3 Mathematical Theory of How (Why) Simple GAs Work	G-9
G.4 Messy Genetic Algorithm (mGA)	G-11
G.4.1 Messy Genetic Algorithm Operators	G-11
G.4.2 Messy Genetic Algorithm Parameters	G-13
G.4.3 Mathematical Theory of How (Why) Messy GAs Work	G-13
G.5 Fast Messy Genetic Algorithm (fmGA)	G-14
G.5.1 Fast Messy Genetic Algorithm Operators	G-14
G.5.2 Fast Messy Genetic Algorithm Parameters	G-15
G.5.3 Mathematical Theory of How (Why) Fast Messy GAs Work	G-15
Appendix H. Data Visualization	H-1
H.1 Introduction	H-1
H.2 Landscape Visualization	H-1

	Page
Appendix I. Random Number Generators	I-1
I.1 Introduction	I-1
I.2 Random Number Generators (RNG)	I-1
I.2.1 Random Number Generator Correlation	I-2
Appendix J. Additional Design and Implementation Information	J-1
J.1 Introduction	J-1
J.2 GA High-Level Design	J-1
J.2.1 Simple Genetic Algorithm	J-2
J.2.2 Messy Genetic Algorithm	J-6
J.2.3 Fast Messy Genetic Algorithm	J-8
J.3 GA Low-Level Design and Implementation	J-13
J.3.1 Simple Genetic Algorithm	J-15
J.3.2 Messy and Fast Messy Genetic Algorithms	J-15
J.4 Genetic Algorithm Fitness Functions for Energy Minimization	J-16
J.4.1 Previous Energy Model Designs	J-16
J.4.2 Energy Model Design Enhancements	J-17
J.4.3 Implementation Details	J-18
J.5 Summary	J-21
J.6 AFIT Implementation of the fmGA PSP Software	J-22
J.6.1 Inputs to AFIT Toolkit	J-22
J.6.2 Outputs from AFIT Toolkit	J-24
Bibliography	BIB-1
Vita	VITA-1

List of Figures

Figure		Page
1.1.	Three Amino Acid Protein	1-5
1.2.	PFP Simplified	1-6
2.1.	The Folding Problem	2-3
2.2.	[Met]-Enkephalin (Primary Form)	2-8
2.3.	Model Polyalanine ₁₄ (Primary Form)	2-8
2.4.	Meta-Stable State	2-11
2.5.	The Glassy Funnel	2-12
2.6.	Stand-off View of the PSP Landscape	2-13
2.7.	Bond and Angle Energy Equations	2-14
2.8.	Torsion Potential	2-14
2.9.	Improper Torsion	2-15
2.10.	Lennard Jones Potential	2-15
2.11.	Hydrogen Bonding Energy Reduction	2-15
2.12.	Water Water Interaction	2-16
2.13.	CHARMm Energy Model	2-16
2.14.	AFIT's CHARMm Energy Model	2-16
2.15.	Partial Double Bonded Characteristics	2-18
2.16.	The trans Formation	2-19
2.17.	The cis Formation	2-20
2.18.	The Original Ramachandran Plot [119]	2-21
2.19.	Stryers Ramachandran Plot [140]	2-22
2.20.	First Transformation on Ramachandran Plot Equation	2-23
2.21.	Applying First Transformation on Ramachandran Plot	2-23
2.22.	Second Transformation Equation	2-24
2.23.	Applying Second Transformation on Ramachandran Plot	2-25

Figure		Page
2.24.	Φ Axis Transformation	2-26
2.25.	Pseudo Algorithm for Messy GAs	2-33
2.26.	Pseudo Algorithm for Fast Messy GAs	2-36
3.1.	AFIT's ToolKit	3-2
3.2.	pfmGA Algorithm Pseudocode	3-11
3.3.	Heterogeneous pfmGA Algorithm Pseudocode	3-12
3.4.	Epoch fmGA Algorithm Pseudocode	3-13
3.5.	Searching For Secondary Structure fmGA Algorithm Pseudocode	3-15
3.6.	Population Angle Analysis	3-16
3.7.	Population Angle Analysis - 1st Search	3-16
3.8.	Population Angle Analysis - 2nd Search	3-17
3.9.	Original Ramachandran Decoding	3-18
3.10.	Deerman's Modified Ramachandran Decoding [25]	3-19
3.11.	CHARMm Energy Model	3-19
3.12.	Sub-component Summation Code of AFIT's CHARMm Energy Fitness Function	3-20
3.13.	Modified Sub-component Summation Code of AFIT's CHARMm Energy Fitness Function	3-21
5.1.	Serial Run with Different Population Sizes	5-2
5.2.	Logarithmic Performance Comparison of the Four Computing Platforms	5-4
5.3.	Speedup Per Number of Nodes	5-8
5.4.	Mean Time to Complete Each Building Block	5-11
5.5.	Fitness Evaluation Calls per Number of Nodes	5-12
5.6.	Worst/Best Fitness Evaluation Results	5-13
5.7.	Fitness vs Building Block Size	5-14
5.8.	Standard Deviation of Runtime for Different Computing Platforms	5-15

Figure		Page
5.9.	Generating Competitive Template vs Sweep (Timing)	5-16
5.10.	Generating Competitive Template vs Sweep (Fitness)	5-17
5.11.	Randomly Generated Points vs Fitness	5-18
5.12.	Randomly Generated Points vs Fitness	5-18
5.13.	Comparison Between Homogeneous and Heterogeneous (Best) Fitness Values	5-22
5.14.	Building Block vs. Fitness Progress	5-27
5.15.	Fitness vs. Epoch	5-30
5.16.	Effects of Short-circuiting the Results with Respect to Time	5-32
5.17.	Kruskal Wallis Comparison Between Serial (Popsiz e = 200) and 8- node (Popsiz e = 800)	5-38
5.18.	Kruskal Wallis Comparison Between Serial (Popsiz e = 200) and 32- node (Popsiz e = 4500)	5-39
5.19.	Comparison Between Ranking Means of Secondary Structure Search Results	5-40
B.1.	4 × 4 Mesh Interconnection Network	B-8
B.2.	Dimension 4 Hypercube Interconnection Network	B-8
D.1.	Protein Bond Length	D-6
D.2.	Protein Bond Angle	D-6
D.3.	Protein Dihedral Angle	D-6
D.4.	The Folding Problem	D-8
D.5.	ECEPP/2 Energy Model as Implemented by AGCT	D-13
D.6.	CHARMM Energy Model as Implemented by AGCT	D-14
E.1.	Pseudo-code for Extra Work Loop	E-3
E.2.	Improved Extra Work Loop	E-4
G.1.	Simple Genetic Algorithm Data Structures and Terminology	G-5

Figure		Page
G.2.	Single-Point Crossover	G-6
G.3.	Bitwise Mutation	G-6
G.4.	Roulette Wheel Selection	G-7
G.5.	Pseudo Algorithm for Simple GAs	G-8
G.6.	Simple Evolutionary Algorithm (GA)	G-8
G.7.	Pseudo Algorithm for Messy GAs	G-12
G.8.	Pseudo Algorithm for Fast Messy GAs	G-15
H.1.	P-norm Equation	H-2
H.2.	P-norm Equation	H-2
H.3.	Visualization Legend	H-3
H.4.	Sample Fitness vs. Time Graph	H-4
I.1.	Repeat of Random Numbers in a Subsequence of Random Numbers	I-3
J.1.	UNITY Description of a Simple Genetic Algorithm	J-3
J.2.	UNITY Description of SGA Initialization	J-3
J.3.	UNITY Description of Roulette-Wheel Selection	J-4
J.4.	UNITY Description of Single Point Crossover	J-4
J.5.	UNITY Description of Bitwise Mutation	J-4
J.6.	UNITY Description of a Messy Genetic Algorithm	J-7
J.7.	UNITY Description of Partially Enumerative Initialization	J-8
J.8.	UNITY Description of mGA Tournament Selection	J-9
J.9.	UNITY Description of Cut and Splice	J-10
J.10.	UNITY Description of Splice	J-10
J.11.	UNITY Description of Cut	J-11
J.12.	UNITY Description of a Fast Messy Genetic Algorithm	J-12
J.13.	UNITY Description of Probabilistically Complete Initialization	J-13
J.14.	UNITY Description of fmGA Tournament Selection	J-14

Figure		Page
J.15.	Incorrect Dependent Dihedral Angle Rotation	J-19
J.16.	Correct Dependent Dihedral Angle Rotation	J-19
J.17.	Dihedral Angle Periodicity	J-20
J.18.	AFIT's ToolKit	J-23
J.19.	Z-matrix File Format	J-23
J.20.	Example lines of Z-matrix File	J-24

List of Tables

Table		Page
1.1.	Practical Differences of the Computational Engines	1-4
1.2.	High Level Design Options	1-8
2.1.	Accepted Tertiary Dihedral Angular Values for the Met-Enkephalin Protein [25]	2-9
2.2.	Accepted Tertiary Dihedral Angular Values for the Model Polyalanine ₁₄ Protein [25]	2-9
2.3.	Comparison of Energy Functions	2-17
2.4.	Bond Angle Constraints	2-19
2.5.	The trans Formation	2-20
2.6.	Loose Constraints for Met-Enkephalin	2-25
2.7.	Loose Constraints for Polyalanine	2-26
3.1.	Available Genetic Algorithm Implementations	3-3
3.2.	Gate's Original Input File for the Serial Version of the fmGA	3-5
3.3.	Merkle's Modified Input File for the Parallelized Version of the fmGA	3-6
3.4.	Secondary Structure and Additional Parallel Input Parameters for the fmGA	3-8
3.5.	[Met]-Enkephalin Encoding Scheme	3-16
3.6.	<i>Polyalanine</i> ₁₄ Encoding Scheme	3-17
4.1.	What to Keep In Mind When Testing?	4-2
4.2.	Professor Tufte's One-day Course on Presenting Data and Information	4-2
4.3.	fmGA Output Data	4-5
4.4.	Parametric Settings for Most Experiments	4-7
4.5.	Input Schedule for the Polyalanine Protein	4-8
4.6.	Input Schedule for the [Met]-Enkephalin Protein	4-8

Table		Page
4.7.	List of Experiments	4-9
4.8.	Possible Experiments Not Conducted During this Research	4-9
5.1.	Serial Run with Varying Population Sizes	5-2
5.2.	Mean Completion Times vs Node Size	5-5
5.3.	Speedup Per Number of Nodes	5-6
5.4.	Speedup Per Number of Nodes	5-8
5.5.	Fitness Evaluation Results Ordered by the <i>n_a</i> Parameter	5-11
5.6.	Fitness Evaluation Results Ordered by Number of Nodes	5-13
5.7.	Energy (kcal) Fitness Values for Random Search With and Without Sweeps	5-19
5.8.	Stochastic Search Speedup	5-20
5.9.	Utilization of Available Computational Time	5-21
5.10.	Protein Energy (kcal) Fitness Values	5-22
5.11.	Secondary Structure Constraint Analysis (in kcal)	5-24
5.12.	Protein Energy (kcal) Fitness Values	5-24
5.13.	Met-Enkephalin Energy (kcal)	5-26
5.14.	Polyalanine ₁₄ Energy (kcal)	5-27
5.15.	Protein Timing and PVE	5-28
5.16.	Epoch Comparison	5-30
5.17.	Results of Ramachandran Constraint Incorporation Into the fmGA	5-33
5.18.	Optimal Solution Angular Values for [Met]-Enkephalin	5-33
5.19.	Optimal Solution Angle Values for Polyalanine ₁₄	5-33
5.20.	Best Dihedral Angular Values for [Met]-Enkephalin	5-34
5.21.	Best Dihedral Angular Values Found for Polyalanine ₁₄	5-34
A.1.	NX Function Calls [65]	A-2
A.2.	MPI Functions [113]	A-4
A.3.	Reasons for Choosing MPI as the Parallel Communications Medium	A-4

Table		Page
B.1.	Cluster of Workstations (Sun Microsystems)	B-2
B.2.	Cluster of Workstations (Sun Microsystems) Cont.	B-3
B.3.	Pile of PCs	B-4
C.1.	Pile of PCs	C-2
D.1.	Abbreviation of Amino Acids [139]	D-2
D.2.	Levels of Protein Structures [139]	D-4
D.3.	ϕ, ψ Pairs of Common Secondary Structures	D-4
D.4.	Enumeration Time of a 1.3×10^{30} Search Space at One Solution per Clock Cycle	D-7
D.5.	Abbreviation of Amino Acids [139]	D-8
D.6.	Time Complexity of Energy Minimization Methods	D-12
F.1.	Gate's Predicted Population Sizes to Minimize the [Met]-Enkephalin based on Goldberg's Population Sizing Equations	F-1
H.1.	Visualization Legend	H-3
J.1.	Energy Component Comparison	J-17

Abstract

The ability to accurately predict a polypeptide's molecular structure given its amino acid sequence is important to numerous scientific, medical, and engineering applications. Studies have been conducted in the application of Genetic Algorithms (GAs) to this problem with promising initial results. In this thesis report, we use the fast messy Genetic Algorithm (fmGA) to attempt to find the minimization of an empirical CHARMM energy model and generation of the associated conformation. Previous work has shown that the fmGA provided favorable results, at least when applied to the pentapeptide [Met]-Enkephalin. We extend these results to a larger Polyalanine₁₄ peptide by utilizing secondary structure information as both searching constraints and seeding the initial population. Additional efforts were conducted to improve the performance of the algorithm with respect to solving the Protein Structure Prediction (PSP) problem through a short-circuiting operator—where complete evaluation of the fitness function is halted if initial results are not promising, and by conducting additional searches on faster machines in a heterogeneous environment. Results indicate that, on average, this localized search tends to produce better final solutions. Finally, the fmGA as applied to the PSP problem is analyzed and shown to have improved performance and effectiveness.

SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH
FAST MESSY GENETIC ALGORITHMS
(SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED
PEPTIDES BY DETECTING SECONDARY STRUCTURES)

I. Introduction

1.1 Protein Structure Prediction: A Grand Challenge Problem

The Protein Structure Prediction (PSP) problem, determining the geometrical conformation of a given protein at its energy minimized state, is one of the grand challenges in microbiology [15] and the Air Force Institute of Technology (AFIT) has had a long-standing interest in attempting to find a "semi-optimal" solution through Evolutionary Algorithmic approaches [4, 96, 6, 25, 40, 41, 5, 43, 42, 88, 89, 44, 92, 91, 94, 40]. Our ability to construct computer algorithms that either simulate or calculate the folding or final conformation has yet to provide a viable answer in an acceptable amount of time. Recently IBM initiated a new project called Blue Gene (see Appendix C for further information) to construct the fastest supercomputer the world has ever seen (Tera speed computations) [122, 121, 120]; however, in this author's opinion even this will prove to be ineffective in helping researchers find optimal solutions in an acceptable time though their efforts will inevitably push forward the research in this and other areas. Even the most powerful of today's super computers, if given a problem of sufficient complexity, would take centuries to return a solution. In fact, it has been estimated that if an exhaustive search of a reasonable discretization of the entire solution space for even the smallest proteins would consume more time than the estimated age of the universe [41]! Yet, if truly complex problems - such as the so-called "Grand Challenges" - are to be solved, improved algorithmic methods must be accompanied by similar improvements in computer technology and IBM's efforts go a long way in improving computational power!

The PSP and Protein Folding Prediction (PFP) problem are two related problems that have the same overall objective: to accurately predict the conformational structure of a protein. The Tertiary Structure Prediction (TSP) approach is primarily concerned with predicting the protein's tertiary structure without regards to how the protein arrived at this folded state. On the other hand, the PFP's primary concern is the transition process the protein undergoes starting from the primary structure and ending in the tertiary structure (*ab initio*). A protein is comprised of amino acids linked together through chemical bonding. The tertiary structure of a protein corresponds to positioning all its amino groups in such a manner that the overall molecule has the lowest energy (i.e., conformational energy). Although the structure of a specific protein, hence the positions of all the atoms, can be accurately determined using currently available methods (x-ray crystallography and nuclear magnetic resonance –see appendices for further details), each of these methods requires several years to obtain reliable results for a single protein [4], and the protein must first be synthesized or isolated. Therefore, a portion of the PSP/PFP community has turned its attention to predicting the conformational structure through the use of computer models and simulations. The intent of this research is to continue that exploration and specifically AFIT's efforts to explore genetic algorithms as a possible means to solve the PSP problem for larger-sized proteins.

1.1.1 PSP Problem Class. The PSP problem belongs to the class of problems for which currently there is no known non-polynomial-time complexity nondeterministic algorithm (i.e., NP). NP-complete problems are the class of problems that have the aforementioned property and are able to map to each other via some polynomial algorithm. [17] Thus by mapping the PSP or PFP problem into a “known” NP-complete problem we can demonstrate equivalency and thus show that the PSP problem is an NP-complete problem. The PSP problem is in NP because the size of the search space is defined by the number of allowed rotational angles raised to the power of the number of independently variable dihedral angles. In other words, the cardinality = $n^{Protein}$; where n is the number of positions a dihedral can hold and the cardinality represents the number of independent variable dihedral angles within the protein. Deerman provides a simple example to demonstrate this and is included here [25].

If we used a three-peptide protein with p_1 rooted at the axis of a normal Cartesian plane (0,0), p_2 can be positioned at 0° to 360° about the origin, and p_3 can similarly rotate about p_2 . In this example, $P = (p_1, p_2, p_3)$ and $n = 360$ yielding 129600 possibilities (360^2).

This simplification allows for more variability within the rotation than what is allowed by nature, which is explained in more detail in the following section. Mapping the PSP problem to another known NP-complete class problem is a trivial but rather lengthy matter. The requirement is to prove that the PSP problem is polynomial transformable to another known NP-complete problem. The argument and proof is in [17]. The fact that the PSP problem is shown to be NP-complete forces us to concede that with today's knowledge and computational capabilities, we will not be able to conclude that "the" optimum solution has been found.

1.1.2 PSP Structure. The structure of the PSP problem is exclusively defined by the means of calculating the conformational energy. There are a number of methods available; for instance, empirical methods take dihedral bond angles into account while holding such energy terms like bond length and bond angle constant and furthermore, they usually don't account for interactions between the protein and the surrounding solvent. On the other hand, the *ab initio* method uses all atomic interactions when calculating the conformational energy as the protein folds. Table 1.1 lists the practical differences between the three general computational methods.

Assuming the use of an empirical computational method, the PSP problem structure reduces to:

Given a protein (P) comprised of a chain of peptides (p), each having a dihedral angle associated on one end, given the position of the first peptide find the positions of (p_1, p_2, \dots, p_n) such that the conformational energy level (C) is the lowest:

$$\exists P : \min \left| \sum_{i=1}^n C(p_i) \right| \quad (1.1)$$

Simplified Assumption

Table 1.1. Practical Differences of the Computational Engines
Empirical Methods

- Used in molecules containing thousands of atoms
- Can be applied to organics, oligonucleotides, peptides, and saccharides (metallo-organics & inorganics in some cases).
- Vacuum, implicit, or explicit solvent environments
- Can only be used to study ground state.
- Can be used to explain thermodynamic and kinetic properties.

Semi-Empirical Methods

- Limited to hundreds of atoms
- Can be applied to organics, organo-metallics, and small oligomers (peptide, nucleotide, saccharides).
- Can be used to study ground, transition, and excited states (certain methods).

Ab Initio Methods

- Limited to tens of atoms and still best performed using a supercomputer.
- Can be applied to organics, organo-metallics, and molecular fragments (e.g. catalytic components of enzyme).
- Can be used to study ground, transition, and excited states (certain methods).

Of course, this is a drastic over simplification of the PSP problem because there are many molecular chemistry concepts that we have not yet taken into account. The earlier assumption that a peptide only has a single dihedral angle is intuitively wrong. Basic geometry tells us that to position an object in 3D space three angles must be used to define rotation about the x, y, and z-axis. Thus, in molecular chemistry, a polypeptide is viewed from the description of a single peptide unit. The polypeptide has a direction associated with it for geometric and scientific description purposes. The peptide begins at the α -amino and ends at the α -carboxyl group (see Figure 1.1). Each peptide has three associated angles: Ψ (psi), Φ (phi), and Ω (omega).

A peptide can be considered rigid and planar about the Ω dihedral angle, but this simplifying assumption still allows each peptide structure to rotate at either side of the α -carbon because these bonds are pure "single" bonds. Rotations about these bonds are

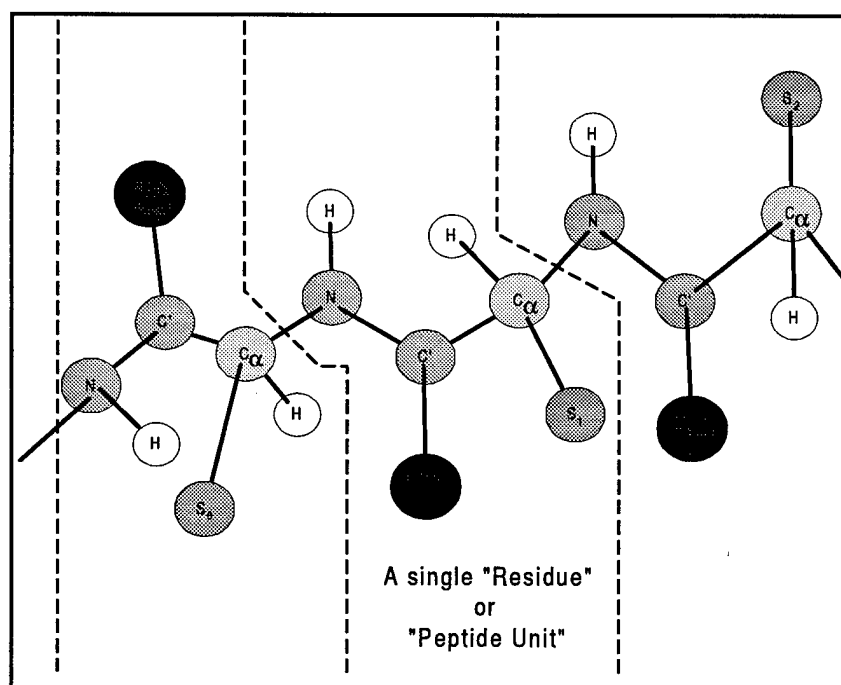


Figure 1.1. Three Amino Acid Protein

defined as Ψ (psi) and Φ (phi) dihedral angles. Looking at Figure 1.2, Ψ refers to the angle of rotation of the plane on the left about the C-C α single bond and Φ refers to the angle of rotation of the plane on the right about the C α -N single bond. [140]

1.2 (Search and Parallel) Algorithms

Recent research efforts have turned towards creating generalized search algorithms which are, more often than not, parallelized to "overpower" the many difficulties associated with the PSP problem. AFIT has been a leader in the pursuit of solving this particular Grand Challenge problem using this particular method and has developed a number of stochastic-based, parallel algorithms in its attempt to find a generalized algorithm that can consistently find a "good" semi-optimal solution. [28, 79, 6, 42, 88, 96, 94, 41, 95, 90, 129].

1.2.1 Search Algorithms. There are two main algorithmic approaches to solving problems (deterministic and stochastic). As eluded to before, it would be impossible with today's hardware/software systems to solve "any" NP-complete problem with any

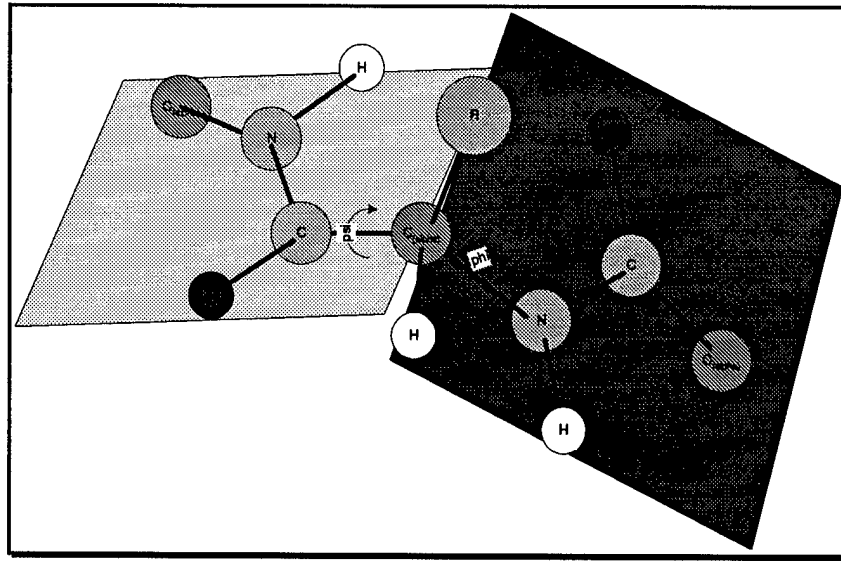


Figure 1.2. PFP Simplified

absolute confidence of finding “the” best solution. As a result, many turn to stochastic-based algorithms to somewhat randomly search the solution space to obtain a semi-optimal solution. It is recognized that there is almost a 100% chance that the “best” answer will not be found, but a semi-optimal solution can be expected with some degree of confidence. There are a number of basic strategies that have been developed with respect to stochastic-based algorithms and are further explained in the appendices. It should be noted that AFIT has concentrated predominately on Genetic Algorithms in finding a “good” algorithmic approach to solving the PSP problem.

1.2.2 Parallel Algorithms. Parallel computing is a field of computer science and engineering that transforms problems traditionally solved sequentially by decomposing them into independent subproblems that can be solved simultaneously on separate processors. Essentially, the intent of parallelizing a given algorithm is to either decrease computational time associated with “running” an algorithm (efficiency or speedup) and/or to improve overall effectiveness with respect to a better solution. It should be noted that it is possible to improve both criteria at the same time; however, at some point, computational speedup benefits are overcome by costs associated with communications between nodes in the parallel environment.

AFIT's use of GAs allowed graduate students to readily parallelize the software and test it in various homogeneous and heterogeneous computing environments. By instantiating a number of copies of the GA, the algorithm can execute on an equal number of nodes and each node can work on a unique set of population members. There are a variety of ways that communication between subpopulations can occur and prior AFIT students have looked at this issue. [91, 41, 92, 93]

A problem is well-parallelized if it can be computed very quickly by an algorithm which uses a feasible amount of processors [142]. A natural way to uncover whether or not a particular algorithm is parallelizable is to determine if it belongs in Nick's Class (NC) [142]. NC contains the class of computational problems that can be solved on the parallel random access machine (PRAM) model by a deterministic algorithm in polylogarithmic time using only a polynomial number of processors. (The PRAM theoretical model of computation is formally defined as a computer consisting of p processors and a global memory of infinite size that is uniformly accessible to all processors). [78] In general, NC includes algorithms that satisfy:

$$P(n) = nl \Rightarrow T_p(n) = \frac{n^l \log^k n}{n^l} = \log^k n \quad (1.2)$$

Mathematical Representation of NC Problem

where, T_s = sequential execution time, T_p = parallel execution time.

Genetic algorithms are one such class of algorithms that satisfy this NC structure. A wide variety of parallel architectures have been designed and implemented. High level design options are shown in Table 1.2. [78]

No single architecture, of course, has been shown to be clearly superior for all applications as this would go against the No Free Lunch Theorem. [147]

1.3 Objectives

The overall purpose of this research effort is to improve upon the original fmGA as applied to the PSP problem in three ways. First, to improve the overall effectiveness of the

Table 1.2. High Level Design Options
Single Instruction, Multiple Data stream (SIMD)

- a single control unit sends instructions to each processing unit

Multiple Instruction, Single Data stream (MISD)

- each processor performs different operations on a single data stream (i.e., vector processors)

Multiple Instruction, Multiple Data stream (MIMD)

- each processor is capable of executing a different program independent of the other processors

algorithm in finding a better energy minimization value; second, to decrease computational time thereby improving overall performance of the algorithm; and finally, to scale the algorithm such that it finds viable solutions to larger protein structures.

1.4 Approach

In order to accomplish these three tasks, the original fmGA PSP code, as modified by Gates and Merkle [41, 88], is analyzed to determine if algorithmic improvements can be found which; if incorporated into the algorithm, improves the algorithms overall performance and effectiveness. In order to scale the algorithm to larger-sized proteins, additional domain space knowledge like Ramachandran constraints [119] and secondary structure information [127, 138] is incorporated into the algorithm. Finally a direct comparison between Gates and Merkle's original work is discussed.

1.5 Assumptions

With respect to this thesis effort there are only three assumptions being made. First, the fmGA algorithm obtained from previous students is correct in its implementation; second, the associated energy minimization values obtained via x-ray crystallography is the most accurate values obtained to date; and, the NX version of the parallelized fmGA is correctly implemented.

1.6 Scope

This thesis discusses the following:

1. PSP problem domain
2. Evolutionary computation—to include both evolutionary algorithms and genetic algorithms in general and the fast messy Genetic Algorithm (fmGA) specifically; stochastic—(of which evolutionary computation falls under) and deterministic-based algorithms
3. Re-parallelization of the fmGA and its performance across three distinct computing platforms
4. Determination of parametric values
5. Incorporation of additional domain space information, such as Ramachandran plots and secondary structure information in both seeding the population and searching for secondary structures
6. Epochs
7. Heterogeneous Implementation of the fmGA

1.7 Layout of the Thesis

Literature Review

The literature review provides the basis on which to build new theories. The review is used to determine the current knowledge in the applicable fields of study. This literature review will focus on evolutionary algorithms, parallel algorithms, and the protein structure prediction problem. The end result of the literature review should be a concise hypothesis stating what this investigation attempts to demonstrate.

Hypothesis

Following the literature review, a hypothesis is formed which the research attempts to validate.

Experiment Design and Implementation

Code design and implementation is required to validate the hypothesis generated in the previous section. The design is built upon work completed by previous AFIT students [4, 88, 41, 43, 25] and attempts to extract empirical evidence to support the hypothesis and establish some theoretical basis which explains the final results. After the design is complete, implementation should be a relatively mechanical process.

Experiment Execution

Even with a well designed set of experiments, the fact that this research attempts to find a viable solution to larger-sized proteins is costly in both time and resources. If previous work [4, 88, 41, 43, 25] is any indication of the runtime requirements, then conducting the experiments requires days as opposed to hours or minutes. Depending on the objective under review, the tests conducted are done on various parallel platforms up to 128 nodes.

Analysis

The final step involves analyzing the data generated from performing the experiments. This information is used to prove or disprove the hypothesis based on the theoretical and empirical evidence gathered in the previous activities. This is the time to stop, evaluate what has been accomplished, and make recommendations on where future research should be focused.

1.8 Conclusion

There exist classes of problems which can't be solved in reasonable time strictly by faster computation and the PSP problem belongs to this class. It has been shown that the PSP problem is of great interest to the scientific community and the United States Air Force. Evolutionary Algorithms, in particular the fmGA, is used to provide semi-optimal solutions to this problem. This thesis, while continuing the AFIT work in GAs applied to the PFP and PSP, attempts to improve the performance and effectiveness of the algorithm through the incorporation of additional domain space information, as well as improving its performance on heterogeneous computing platforms.

This chapter has outlined the general problem, described the main components, rationalized the need to expend research effort on genetic algorithms and the protein folding problem. The next chapter surveys existing literature in the fields of genetic algorithms and Protein Structure Prediction problem.

II. Background: Literary Review and Requirements

2.1 Introduction

The protein structure problem (PSP) is a National Grand Challenge problem in biochemistry and high-performance computing [15]. This thesis effort addresses tertiary structure prediction (TSP), which is one approach to solving the PSP. The challenge of the TSP is to find a method to predict the three-dimensional structure of a protein based strictly on its amino acid structure.

A TSP solution enables the *evaluation* of many proteins in a search for one with a specific property or function. Possible applications include the development of pharmaceuticals with few or no side effects, energy conversion and storage capabilities (similar to photosynthesis), biological and chemical catalysts and regulators, angstrom scale information storage, and possible optical/chemical shielding from harmful radiation sources [11, 82, 114].

This chapter covers the following topics in the order presented in the following list:

1. Protein Structure Problem
2. Introduction to Proteins and Associated Terminology
3. Landscape Visualization
4. Search Algorithms (Deterministic and Stochastic)
5. Parallel Architectures
6. Random Number Generators

It is not the intention of this thesis to be all-inclusive; in fact, a number of references are provided for the reader if he or she wishes to delve further into various areas of this research.

2.2 The Protein Structure Problem

This section provides the background required to understand the mechanics and ramifications of the PSP. Subsection 2.2.1 defines terminology in the biochemistry domain.

Subsection 2.2.2 describes the expensive experimental techniques used to determine the structure of proteins. Finally, Subsection 2.2.3 examines various models used to predict the structure of polypeptides and proteins.

2.2.1 Introduction to Proteins and Associated Terminology. Proteins (polypeptides) are linear sequences of the 20 naturally occurring amino acids (see Appendix D). Each amino acid consists primarily of three common *backbone* atoms (a nitrogen and two carbons [N-C_α-C_γ]) and a distinct combination of atoms and covalent bonds, called the *side-chain* (S_i), connected to the C_α carbon atom. A particular protein is defined by a unique amino acid sequence known as the *primary structure* of the protein [11, 82, 81]. As the amino acids form into proteins via peptide bonds, they give up a water molecule. These linked amino acids are called *residues*. The primary structures of more than 50,000 proteins are currently known and is expected to continually increase in the foreseeable future strictly based on results obtained from the Human Genome Project and the ease with which sequences are experimentally determined [82, 111].

Subsequences of proteins tend to exhibit regular patterns and two of the most common patterns are the *α-helices* and *β-sheets* (see Appendix D). These two patterns describe the *secondary structure* of a protein [11]. Some researchers are investigating the utility of predicting secondary structure as the first step of tertiary structure prediction [81]. This technique has had limited success in the past; however, recently researchers have been able to obtain up to 75% accuracy in determining the secondary structure of a specific sequence of amino acids. [18, 68, 103, 14, 125, 98, 138, 128] The problem is that even though certain residues are found more frequently in specific secondary structure, the greatest preference is only twice that of other secondary structures. In most cases, the preference is much smaller [140].

The three-dimensional structure of a protein, called the *tertiary structure* or *conformation* of the protein, is the major determinant of its function. Proteins assume their *native* conformation, which is unique and compact, in their natural biological environment (typically in aqueous solution, at neutral pH and 20–40° C) [11, 82]. A protein in its native conformation is only slightly more stable than the various conformations with

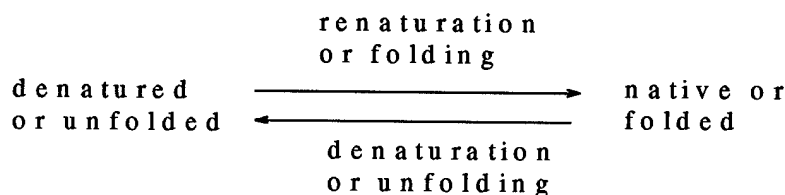


Figure 2.1. The Folding Problem

marginally higher energies. Normally, there is only a 10 kcal/mol energy difference between the completely folded and unfolded conformations. [11, 82, 81]. Although complex, the actual concept is relatively simple in that a fully extended protein folds to its native state through a renaturation or folding process with the opposite process taking a natively conformed protein and extending or unfolding it. This is called denaturation or unfolding (see Figure 2.1 for a visual representation). [16]

Using the internal coordinate system, the position of all atoms in a protein can be determined given the position of one atom, the *bond length* of each covalently bonded pair of atoms, the *bond angle* formed by each triplet of bonded atoms, and the *dihedral* or *torsional angle* formed by each bonded group of four atoms. Every protein has $3n - 6$ degrees of freedom (roughly speaking, number of variables) where n is the number of atoms. However, the bonds and bond angles are relatively rigid, therefore the independent dihedral angles are left as the only dominant factor to determine the tertiary structure of a protein [11, 81]. Each amino acid contains a ϕ , ψ , and ω dihedral angle along its backbone and zero or more χ_i dihedral angles along its side chain.

If we discretize the domain of the dihedral angles so that there are d possible values, then the size of the search space is given by d^N where d is the number of independently variable dihedral angles. AFIT's implementation of this results in a $.3515625^\circ$ discretization of the $0-360^\circ$ dihedral angle domain and for a small protein with 24 independently variable dihedral angles (like the Met-Enkephalin peptide), the search space contains $1024^{24} \approx 1.767 \times 10^{72}$ conformations. Even moving to a slightly larger protein such as a model

Polyalanine of 14 residues results in a search space of nearly $1024^{56} \approx 3.77 \times 10^{168}$. If we consider the fact that there exists proteins that are literally thousands of residues in length then the ability to find the natural conformation of that protein would require researchers to prune off a very significant portion of the search space.

2.2.2 Experimental Tertiary Structure Determination. In comparison with the number of known protein sequences, the number of known native conformations is extremely small (approximately 400). [41] The tertiary structures of these proteins have been determined experimentally using either X-ray crystallography or nuclear-magnetic-resonance (NMR) spectroscopy. These techniques are inadequate for the task because they take one to two years to obtain results for a single protein [11, 82]. Thus, the quest is on to find a method to accurately predict the structure without actually observing it.

2.2.3 Tertiary Structure Prediction (TSP). The ability to accurately predict a polypeptide's molecular structure given its amino acid sequence is important to numerous scientific, medical, and engineering applications. Studies have been conducted in the application of Genetic Algorithms (GAs) to this problem with initial results shown to be promising. In this thesis we use the fast messy Genetic Algorithm (fmGA) to attempt to find the minimization of an empirical CHARMM energy model and generation of the associated conformation. Previous work has shown that the fmGA provided favorable results, at least when applied to the pentapeptide [Met]-Enkephalin. This thesis effort attempts to extend the favorable results to a larger protein by incorporating additional secondary structure information into the algorithm. This information is utilized to conduct localized searches on the energy landscape, restrict the search space and initialize the initial population with potentially good building blocks.

To reduce the gap between the number of known protein sequences and native conformations, we need to be able to reliably predict the tertiary structure of proteins in a reasonable amount of time. *Exact* versions of the classical methods discussed next are theoretically capable of finding the native tertiary structure of any protein. In practice, the computational cost of the calculations prohibits the use of these exact methods. The classical methods that are computationally viable are typically relaxed formulations that

ignore the high-order interaction terms. The applicability of the other prediction methods discussed below is severely limited.

2.2.4 Classical Prediction Methods. *Molecular dynamics* is a technique that attempts to simulate the protein folding process. The protein is treated as an N-body simulation and Newton's motion equations are solved to determine the location of all the atoms at discrete points in time. Molecular dynamics faces two major difficulties in its attempt to fold proteins. First, the number of atoms that must be simulated is very large:

1. Small proteins contain hundreds of atoms.
2. Larger proteins can be composed of several tens of thousands of atoms.
3. Thousands of atoms must be added to simulate the surrounding solution.

Second, the thermal oscillations of bonded atoms have a period between 10^{-14} – 10^{-13} seconds. Simulation time steps in the femtosecond (10^{-15} sec) range are required to accurately account for these harmonics. These two factors have limited molecular dynamics simulations to less than a few nanoseconds (10^{-9} sec), even on today's fastest supercomputers. That time-frame is ten orders of magnitude (10^{10}) too short to simulate the folding process of most proteins [11, 82]. Using an *extended-atom representation* is one method that can greatly reduce the impact of these two problems. The extended-atom representation combines hydrogen atoms with the heavier atoms they are bonded to. This representation generally halves the number of "atoms" in the problem and allows the size of the simulation time steps to be increased. [7] The *energy minimization* approach assumes that proteins, like other physical systems, assume that state which minimizes total energy in the system; however, this assumption is not universally accepted. [67] There are three types of energy minimization methods that differ by their time complexity and the accuracy of their calculations. *Ab initio* methods calculate the energy exactly. *Semi-empirical* methods eliminate the non-dominating interaction integrals from the calculation. *Force-field* methods simply account for the pairwise interactions between atoms with appropriate parameterizations that implicitly accounts for multi-particle interactions. [82]

CHARMm, AMBER, and ECEPP are three examples of force field energy models. See Figure 2.3 for a comparison between each of these models and how AFIT's implementation of the CHARMm energy model fits in. All atoms with more than three bonds

separating them are considered non-bonded. All three models use the non-bonded term explicitly. ECEPP only calculates bonded and bond angle terms around disulfide bridges and dihedral terms for the independently variable dihedral angles. The CHARMM implementation has additional constraints on the range over which terms are computed. AMBER adds a partial non-bonded calculation to the dihedral term plus an extra term to handle polar hydrogen non-bonded interactions with nitrogen and oxygen [81, 7].

2.2.5 Other Prediction Methods. Structure prediction by *homology* attempts to align the sequence of a protein with an unknown tertiary structure with one whose native conformation is known [82]. It has been observed that if the sequences are similar, then the conformations are nearly identical. An extension of homology, called *sequence-structure alignment*, builds a partial monotonic mapping directly from the sequence of the unknown protein to the known tertiary structure of the similar protein. The differences between the two structures are usually surface characteristics built upon the same core structure. Both of these methods are severely limited by our tiny database of currently known protein structures. They are also incapable of predicting the native conformation of proteins with novel structures such as those the sponsor of this research wishes to examine [82]. However, even with these problems, both methods are being employed as a precursor to solving the tertiary structure problem by first attempting to discern if a secondary structure exists or not. If one is *thought* to exist, then the solution space to be searched is constrained accordingly.

Simplification techniques reduce the conformational search space to a size feasible for today's algorithmic search strategies [11]. *Lattice* models reduce three-dimensional space to a structured grid, where atoms can only be placed on the grid points. The grid is designed to accommodate the typical connections observed in real proteins. Further simplifications have been obtained by reducing or eliminating the explicit representation of side-chains [82].

The determination of the tertiary structure of proteins is a major challenge in biochemistry. Experimental techniques are considered accurate but time consuming, and are incapable of keeping pace with the number of protein sequences being discovered. Predic-

tion techniques are hampered by the size of the conformational search space and the time complexity of calculating energy or solving motion equations. However, classical prediction methods, combined with novel search and optimization algorithms, show great potential for both a solution to the PSP and a better understanding of the underlying behavior and operation of biological systems.

The PFP predicts the path taken by a protein (or polypeptide) transitioning from an unfolded (or denatured) state to its folded (or native) state. [16] The elementary form of the folding reaction is deceptively simple (see Figure 2.1). A solution to the protein-folding problem has eluded researchers for more than 30 years. [81] Tertiary Structure Prediction (TSP) is a related problem. The essence of the TSP is to predict the compact, three-dimensional shape of a protein as it would exist in nature. This native conformation corresponds to the native state in the PFP. The native conformation determines the protein's biological functions.

2.2.6 Met-Enkephalin and Polyalanine. The protein used in most of the AFIT studies is the pentapeptide [Met]-Enkephalin (see Figure 2.2). It is a relatively small and simple protein defined by the five-amino-acid sequence Tyr-Gly-Gly-Phe-Met using neutral NH₂ and -COOH as terminators at the α -amino and α -carboxyl ends, respectively. It is principally composed of carbon (C), oxygen (O), and nitrogen (N) atoms. The two principal factors influencing the selection of this particular protein for study are; first, its unique and compact natural, biological state (native conformation) is known, and secondly, other researchers have used energy minimization to predict its tertiary structure. [43, 88, 25, 28, 69, 41]

The second molecule, Polyalanine₁₄, was chosen because of its affinity to nicely fold into a α -helical structure. Polyalanine₁₄, a larger polypeptide than [Met]-Enkephalin, is defined by 14 amino-acid groups: *Ala*₁-*Ala*₂-*Ala*₃-...-*Ala*₁₄. We have chosen to use the same end groups as the [Met]-Enkephalin molecule. Figure 2.2 and Figure 2.3 are representation of [Met]-Enkephalin and Polyalanine₁₄, respectively. The figures are labeled to distinguish the dihedral angles along their molecular backbone.

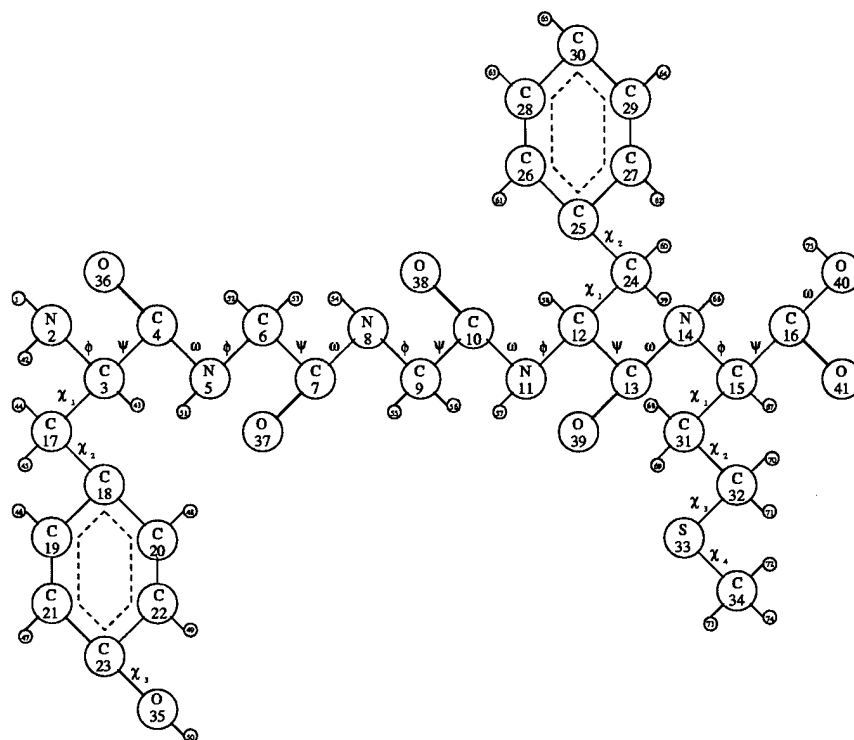


Figure 2.2. [Met]-Enkephalin (Primary Form)

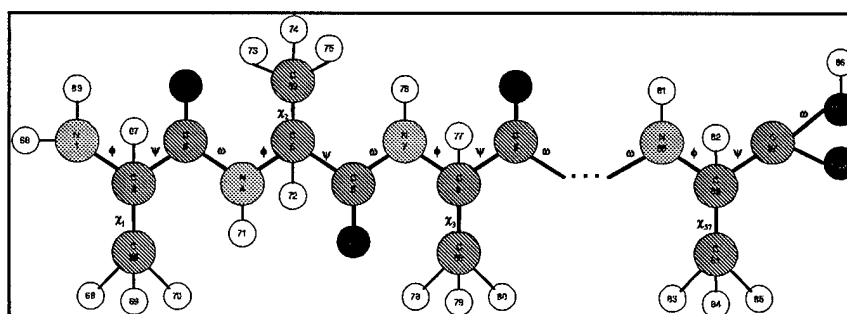


Figure 2.3. Model Polyalanine₁₄ (Primary Form)

Table 2.1. Accepted Tertiary Dihedral Angular Values for the Met-Enkephalin Protein [25]

Residue	Dihedral Angle (degrees)						
	Φ	Ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr	-86	156	-177	-173	79	166	—
Gly	-154	83	169	—	—	—	—
Gly	84	-74	170	—	—	—	—
Phe	-137	19	-174	59	-85	—	—
Met	-164	160	-180	53	175	-180	-59

Table 2.2. Accepted Tertiary Dihedral Angular Values for the Model Polyalanine₁₄ Protein [25]

Residue	Dihedral Angle (degrees)						
	Φ	Ψ	ω	χ_1			
Ala ₁	65°	(30°–35°)	180°	60	-60	120	
Ala ₂	65°	(30°–35°)	180°	60	-60	120	
Ala ₃	65°	(30°–35°)	180°	60	-60	120	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	
Ala ₅₇	65°	(30°–35°)	180°	60	-60	120	

Tables 2.1 and 2.2 outline the “correct” dihedral angles value for the “accepted ” energy minimum defined by QUANTA. The conformation energy for [Met]-Enkephalin is -29.225. Alternative molecules have been considered, (e.g., Crambin [132], P27-4, P27-6, and P27-7 [118], and others mentioned in Deerman’s thesis [25]); however, they were not used as they were are considerably larger than the [Met]-Enkephalin and Polyalanine₁₄ molecules. They also have no accepted minimum conformation at this time and additional steps to convert the source files to a standard that can be used by AFIT’s CHARMM implementation is under review and was not readily available.

2.2.7 PSP Landscape. The most challenging aspect of the PSP problem is that the conformation energy calculation creates an enormous number of local minima. Therefore, any attempt using local minimization techniques usually becomes caught in arbitrary local minima. These minima can be arbitrarily far from the global minimum. The small differences between the assumed conformation energy and these minima makes it extremely difficult to know how close one is to the accepted energy minimum simply by comparing to the calculated energy. It is assumed that the protein’s geometry defined by the naturally occurring conformation is the global minimum. [107] Note that the energy model used

and the refinement of the input data required in the model define the size of the energy landscape (i.e., the search space). For example, let's assume an energy model that only requires as input the dihedral angles of a protein that consists of 56 dihedral angles. If each dihedral angle were allowed to rotate freely about each bond without considering any constraints, then N^d , where d is the number of independently variable dihedral angles, would model the size of the conformation space. For argument's sake, we allow two atoms to occupy the same space and let the energy model indicate the invalidity of the resulting protein. Therefore, supposing that each dihedral angle has 360 possible values, our example protein would have approximately 3.77×10^{168} different orientations. If only one tenth of these orientations belonged to the set of possible minima, then there would be 1 out of 3.77×10^{168} chances of randomly finding the global minimum.

The search space terrain of such a protein is extremely rugged consisting of a very large number of valleys and peaks. How a protein, with no known memory capability finds the global minimum in this complex landscape is still a mystery. The process is driven by forces of physics yet to be understood! [74, 107] Experiments suggest that a protein's approach to the global minimum is characterized by two phases. The first phase is a rapid folding phase that results in a nearly folded protein. This is followed by a lag phase which completes the folding process. [107, 74] This suggests the existence of a large energy barrier with many saddles around the valley containing the global minimum (see Figure 2.4 for a pictorial representation of this phenomena). [107]

Other scientists argue that the conformation state might be a metastable state with high barriers, or it might just be the lowest local minimum that is kinetically accessible from most of the protein's energy space. [107] These landscape descriptions allude to the possibility that a naturally occurring protein may not reach its global minimum energy conformation. This is supported by Kaiser's experiments that uncovered a conformation of [Met]-Enkephalin with a lower CHARMM energy value than previously encountered. [70]

The most promising fitness landscape description is referred to as the glassy behavior. [107] (See Figure 2.5 by [8].) The glassy behavior is defined by the situation when the naturally folded state corresponds to a more extended region in the search space where

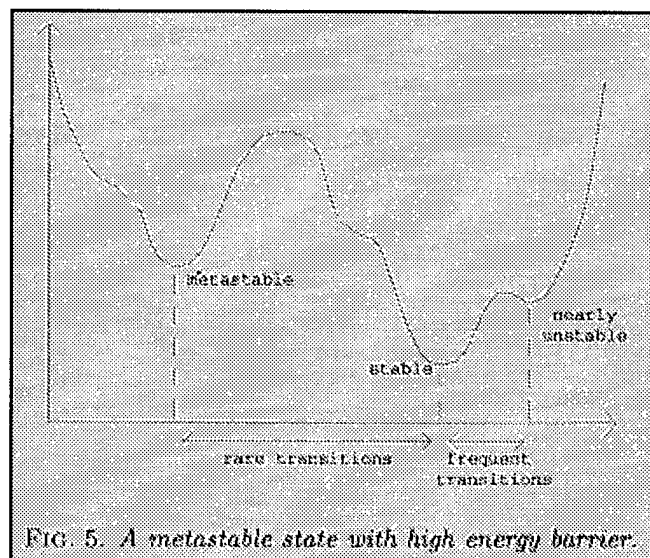


Figure 2.4. Meta-Stable State

there are many closely located minima of approximately the same energy. The differences between the global minimum geometric structure and these false conformations are beyond our current scientific limits to measure. Furthermore, when one of these false conformations is entered into our chosen energy model, the resulting energy can differ from the naturally occurring conformation energy by only a few kilocalories (kcal). Normally, there is only 10 kcal/mol difference between the completely folded and unfolded conformation. [69]

The glassy funnel landscape model combined with the experimental data of rapid initial folding followed by a lengthy lag time to reach the global minimum energy state explains the Levinthal paradox which has confounded researchers for years. The Levinthal paradox simply states that "the time a protein needs to fold is by far not large enough to explore even a tiny fraction of all the local minima believed to comprise the fitness landscape". [69, 107] On the other hand, "when the slope towards the native conformation is dominant over the ruggedness of the landscape, folding kinetics is exponential and [therefore very] fast." [107] This insight allows us to picture the protein quickly folding to an orientation near the native conformation - i.e., the initial rapid folding period. Then, if we imagine this "near orientation" resting on a relatively smooth valley floor, "the lengthy lag

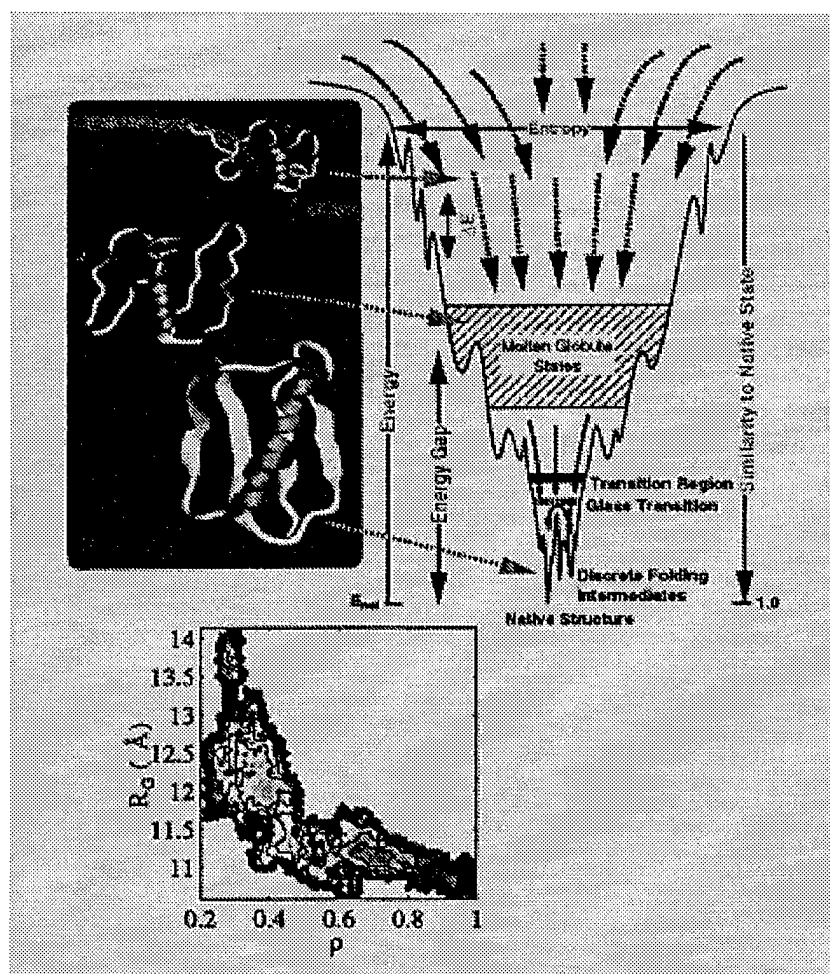


Figure 2.5. The Glassy Funnel

period until the protein 'finds' its native conformation" can be conjectured as the protein searching this small area.

Finally, another aspect that hasn't been mentioned is the fact that for any given protein there exists a large number of local minima valleys, each containing a local minima point. However the point that needs to be made is the overall search space far exceeds the number of local minima valleys by a very large factor. For instance, take the case with the [Met]-Enkephalin, it has been reported elsewhere that there exists approximately 10^6 local minima valleys while the entire search space is somewhere on the order of 1.767^{72} —at least as defined and constructed by prior AFIT graduate students [41, 88, 4, 69, 25]. Given this case, a bird's eye view of the landscape would not be like the microscopic valley view

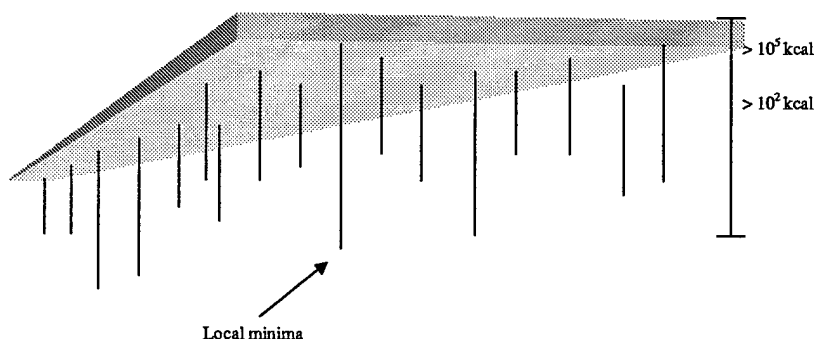


Figure 2.6. Stand-off View of the PSP Landscape

demonstrated in Figure 2.5, but something like a table top with a series of lines that drop perpendicular to the bottom of the table (see Figure 2.6). In an attempt to demonstrate this, a series of tests are run and discussed in a subsequent chapter.

2.2.8 PSP Energy Models. In order to understand and manipulate proteins, we must be able to reliably predict the tertiary structure of the protein in a reasonable amount of time. Generally, there are three different methods to uncover the conformation state of a protein: X-ray Crystallography, Nuclear Magnetic Resonance, or Computational Models. X-ray crystallography and nuclear magnetic resonance spectroscopy are direct methods of measuring the position of each atom within a protein. These methods are rather time consuming (at least two months with current technology), subject to error, and can be laborious! Computational modeling, on the other hand, is somewhat less time consuming and easier to conduct, but these methods are approximations and may not precisely reflect the native structure of a particular protein. Although computational modeling has many shortcomings, it is still an area of utmost interest to biochemists because this form of calculating the native structure provides the greatest possibility of shortening the gap between the discovery or design of a new protein and learning its conformational structure. Appendix D provides an overview of x-ray crystallography and nuclear magnetic resonance spectroscopy, and an in-depth look at the different forms of computational modeling.

The particular computational model we are interested in is the CHARMM (Chemistry at HARvard using Molecular Mechanics) energy model. CHARMM, developed principally

$$E_b = \sum_{bonds} k_b (r - r_0)^2$$

$$E_{\Theta} = \sum_{angles} k_{\Theta} (\Theta - \Theta_0)^2$$

Figure 2.7. Bond and Angle Energy Equations

$$E_{\Phi} = \sum_{dihedrals} |k_{\Phi}| - k_{\Phi} \cos(n_{\Phi})$$

Figure 2.8. Torsion Potential

by Brooks and Bruccoleri [7], is an empirical energy function used in the investigation of the physical properties of a wide variety of molecules. The model is executed on a molecule at a particular temperature in a particular solvent (usually water). The model is based on separable internal coordinates and pairwise non-bonded interaction terms. [7] The model is a composite sum of several molecular mechanics equations. Each is decomposed into its terms in the following series of equations:

Figure 2.7's equation accounts for bond and angle deformations which in most cases at ordinary temperatures and in the absence of chemical reactions are sufficiently small for the harmonic approximation to apply. [7]

The torsion energy term, Figure 2.8's equation, is a four atom term based on the dihedral angle about the axis defined by the middle pair of atoms. For this term the energy constant can be negative (indicating a maximum at the cis conformation), and there may be several contributions with different k_{Φ} and different periodicity's for a given set of four atoms [7].

The Improper Torsion term (Figure 2.9) was developed to maintain chirality about a tetrahedral extended heavy atom, and to maintain planarity about certain planar atoms with a quadratic distortion potential. Without this term, out-of-plane potentials tend to

$$E_w = \sum_{\text{improper, dihedrals}} k_\omega (\omega - \omega_o)^2$$

Figure 2.9. Improper Torsion

$$E_{LJP} = \sum_{\text{pairs, } i \neq j} \left[\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} + \frac{q_i q_j}{\epsilon r_{ij}} \right]$$

Figure 2.10. Lennard Jones Potential

be quadratic. In addition, the term provides a better force field near the minimum energy geometry. [7]

The Lennard-Jones Potential equation (Figure 2.10) accounts for the van der Waals forces of attraction and repulsion energy (the A_{ij} and B_{ij} terms) and the electrostatic attraction and repulsion energy [7]. This equation is the major contributor to the overall energy calculation.

Figure 2.11's equation accounts for a reduction in the van der Waals term between the hydrogen atom and the acceptor atom. [7]

The two equations in Figure 2.12 account for water-water interaction when manipulating the solute in a water solvent. The distance "constraints" (atomic harmonic constraints) are used primarily to avoid large displacements of atoms when minimizing, while still allowing the structure to relax. The dihedral angle "constraints" are used to maintain certain local conformation or when a series of different conformations need to be examined in making potential energy maps. [7]

The CHARMM model is almost a verbatim implementation of the equation in Figure 2.13. The terms k_b and r_0 , k_Q and Q_0 , k_F and n_F , A_{ij} , and B_{ij} are empirical constants

$$E_h = \sum \left[\frac{A'}{r_{AD}^{12}} - \frac{B'}{r_{AD}^6} \right] \cos^n(\Theta_{A-B-D}) \times \cos^n(\Theta_{AA'-A-H})$$

Figure 2.11. Hydrogen Bonding Energy Reduction

$$\begin{aligned} \text{Distance Constraints: } E_{cl} &= \sum K_i (r_i - r_{i0})^2 \\ \text{Dihedral Angle Constraints:} \\ E_{ct} &= \sum K_j (\phi_j - \phi_{j0})^2 \end{aligned}$$

Figure 2.12. Water Water Interaction

$$E_{total} = E_b + E_\theta + E_\phi + E_\alpha + E_{clw} + E_{cl} + E_{td} + E_{ct} + E_{ctd}$$

Figure 2.13. CHARMM Energy Model

supplied as input. These parameters are calculated from "known" protein conformations supplied by the Brookhaven Protein Database (the official repository of protein structures) operated by the National Institute of Health. The number of bonded atoms, the number of atoms forming bond angles, atoms forming dihedral angles, and non-bonded atoms are determined based on the molecule supplied to the model and can be distinguished prior to model execution. The AFIT implementation of the CHARMM model does not account for each of these terms. In the original implementation by Brinkman [4] and the later revision by Gates [41], the hydrogen bonding reduction and water-water interaction terms are excluded because they do not significantly contribute to the overall molecular energy. Therefore, the AFIT implementation does not completely model the molecular interactions (see Figure 2.14), and we can imagine AFIT's CHARMM implementation as modeling the molecular interactions in a vacuum. This is, of course, a common way to calculate the protein structure's energy.

Furthermore, Gates indicated "other" errors in the primary implementation, and he corrected them in order to ensure AFIT's model corresponded with the QUANTATM software package by the addition of the energy constant. [41] CHARMM was chosen as our energy function because it models the most contributing factors as compared to the other commercially available empirical energy function. Table 2.3 provides a comparison between several currently available empirical energy functions and the energy terms they calculate.

$$E_{total} = E_b + E_\theta + E_\phi + E_\alpha + E_{td} + \text{energycons}$$

Figure 2.14. AFIT's CHARMM Energy Model

Table 2.3. Comparison of Energy Functions

Initials	Name	E_b	E_ϕ	E_ψ	E_w	$E_{\text{non-bonded}}$	E_{el}	E_{hb}	E_{cr}	$E_{\text{c}\Phi}$
CHARMm	Chemistry at Harvard using Molecular Mechanics	X	X	X	X	X	X	X	X	X
AFIT CHARMm		X	X	X	X	X	X			
Amber	Assisted Model Building with Energy Refinement	X	X	X		X	X	X		
ECEPP/3				X		X	X	X		
OPLS	Optimized Potentials for Liquid Simulations			X		X	X			

As the number of energy terms included within the model increases, the corresponding complexity/ruggedness of the protein energy landscape also tends to increase. The energy models listed in Table 2.3 are in order by decreasing complexity of the energy landscape (e.g., CHARMm models the most complex landscape of those energy models listed). The smoother the landscape, the less complex and time consuming it is to calculate the protein structure's energy because fewer terms are included. Of course, the calculated energy tends to be less accurate.

2.2.9 Bounding the Search Space. Kaiser's work greatly influenced our efforts at constraining the search space of the PSP problem. As we know, enumerating the whole (discretized) search space is an intractable problem. Therefore, if there were any "generic" way to limit the search space, it would be beneficial to incorporate into our algorithm. Kaiser covers the basics of the geometry found within the backbone of a polypeptide and briefly discusses Ramachandran Plots. [69] But to fully understand Ramachandran's work, we must start by defining a peptide unit. The peptide unit is a rigid planar array of four atoms: nitrogen, hydrogen, carbon, and oxygen. [140] The peptide unit is considered rigid and planar because the bond between the carbonyl carbon (referred to as either C_γ or C') and the nitrogen atom is not free to rotate. This bond has partial double-bond characteristics. [140] (See Figure 2.15)

Several peptides joined together by purely covalent bonding form a chain called a polypeptide. [140] The complete chain of peptide units define the backbone of a protein, and once a polypeptide backbone is configured with its appropriate side chain(s), it is commonly considered a protein. Rotations about the bonds within the protein are described as torsion or dihedral angles that are usually taken to lie between -180° and $+180^\circ$. [140, 16] There are three distinct types of dihedral angles within the protein's backbone. Table ?? lists how they are commonly referenced. Using these conventions, a protein can be characterized as being in either the trans or cis position. The trans position refers to when each of the omega (Ω) dihedral angles assumes a 180° orientation [140, 16]; on the other hand, the cis formation is characterized as the w's assuming an 0° orientation. The trans polypeptide form is naturally favored over the cis formation by approximately 1000:1, because, in the cis form, the $C\alpha$ atoms and the side chains of the neighboring residue are in too close of proximity. [16] The closely positioned side chains greatly influence the pairwise atom interaction energy calculated by the repulsion term in the van der Waals equation in Figure 2.10. This term becomes vary large when the distance between the atoms involved becomes slightly less then the sum of their contact radii. [140] Figure 2.16 and Figure 2.17 illustrate the trans and cis formations. [16]

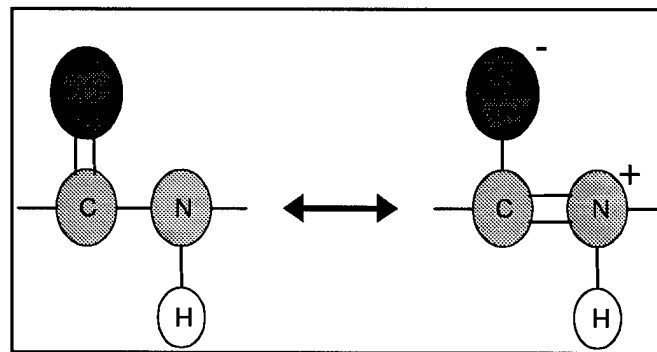


Figure 2.15. Partial Double Bonded Characteristics

Therefore, if we assume the ω dihedral angle is held in the trans position, then the phi ϕ and psi ψ dihedral angles define the backbone of a polypeptide. [69] Allowing slight deviations from the polypeptide's planarity of either the trans or cis conformation, by allowing the w angle to deviate by -20° to $+10^\circ$, is thought to be only marginally

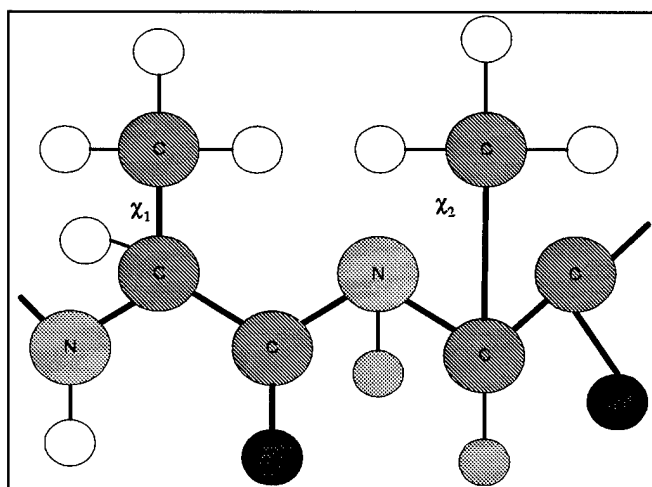


Figure 2.16. The trans Formation

unfavorable energetically in most peptide bonds. [16] Thereby, all three of the angles are responsible for defining the correct folded state of a large molecule. Ramachandran *et al* developed constraints for allowable configurations for polypeptides based upon his two-parameter convention. Ramachandran proposed that it was possible to rotate around the N- α C and the α C-C γ when the groups were linked at the α C atom. [119] Consequently, the relative configuration of two peptide units about the α C atom are specified by just two parameters which he called ϕ and ψ . [119] (See Table 2.4 for translation.)

Table 2.4. Bond Angle Constraints

Bond	Ramachandran	Standard
N- α C	ϕ	Φ (Phi)
α C-C'	ϕ'	ψ (Psi)
C'-N	—	ω (Omega)

The complete configuration of a polypeptide chain is fully specified when each of the α C's parameters (ϕ , ψ) are known. [119] Furthermore, Ramachandran developed a set of allowable regions for these parameters based upon his choice of permissible van der Waals contact distances using a hard sphere model of the atoms and fixed geometries of the bonds. [119, 16] Table 2.5 has a comparison of permissible van der Waals distances as defined by Ramachandran and by Stryer. [140, 119] Ramachandran indicated that two sets of bounds were possible. These bounds, termed "normally allowed" and "outer limit,"

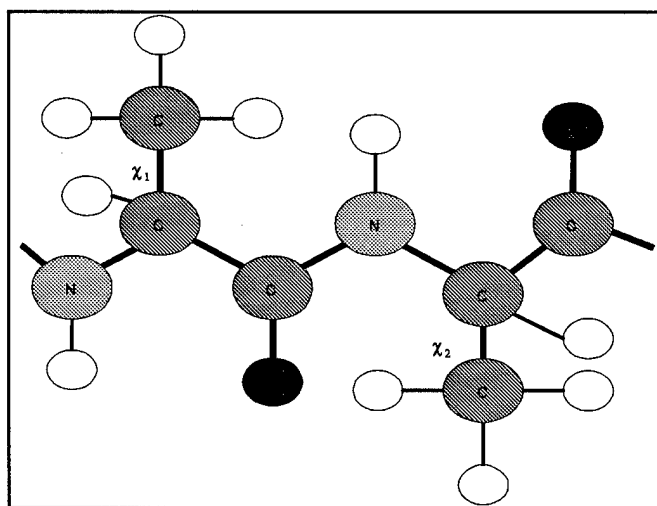


Figure 2.17. The cis Formation

were derived from a detailed analysis of available structural data including amino acids and peptides. [119]

Table 2.5. The trans Formation

Contact	Ramachandran		Stryer Radii (Å)
	Normally Allowed (Å)	Outer Limit (Å)	
C ... C	3.20	3.00	4.0
C ... O	2.80	2.70	3.4
C ... N	2.90	2.80	3.5
C ... H	2.40	2.20	3.2
O ... O	2.80	2.70	2.8
O ... N	2.70	2.60	2.9
O ... H	2.40	2.20	2.6
N ... N	2.70	2.60	3.0
N ... H	2.40	2.20	2.7
H ... H	2.00	1.90	2.4

Based upon his steric constraints, “the permitted ranges for (Φ, Ψ) were obtained, shown in Figure 2.18 [119], corresponding to an angle of 110° between the $N_0-\alpha C_1$ and $\alpha C_1-C_{\gamma 1}$ at the α -carbon atom. [119] “When we allow this angle to vary slightly from 105° to 115° , the allowed regions are altered slightly [119]”.

AFIT’s implementation of the Ramachandran plots for binary based encoding was accomplished by Deerman [25]. According to Deerman, his work is heavily based upon the work and results of Kaiser and his implementation of the Ramachandran Plot in his thesis [69]. According to Deerman, Kaiser’s constrained-GA made use of an existing GA package,

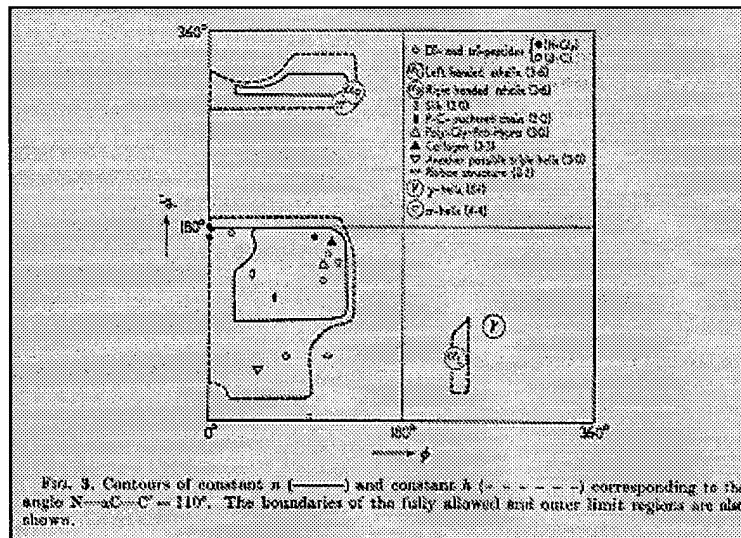


Figure 2.18. The Original Ramachandran Plot [119]

GENOCOP III. He incorporated his constraint system directly into the GA's manipulation of the chromosome [69], but his constrained-GA could not generate an initial population of 50 members using his defined feasible solution space because the feasible search space is much smaller than the entire search space. [69, 70] Therefore, Kaiser had to use a hand picked initial population. It is commonly understood that any constraints placed upon a GA hampers its execution time. Normally, GAs have two choices when they encounter "disallowed" chromosomes: 1) excluded and replaced, or 2) repair the chromosome. If the disallowed chromosome is excluded and replaced, we may find that the GA spends an overwhelming amount of time finding "allowable" chromosomes, depending upon the ratio between the "allowable" search space and the "complete" search space. On the other hand, if we repair every "disallowed" chromosome, the GA must first recognize "why" the chromosome is not allowed and then repair the particular gene(s) in violation. This operation usually overwhelms the GA because it now must have problem domain information embedded within its algorithm. Summarizing Kaiser's work leads to the conclusion that constraints on the search space are "good," but his implementation lead to preconvergence and "islands" of feasible solutions that didn't allow his GA to traverse the search space to find the optimum solution. [69] What we propose is a better way to overcome the preconvergence and "islands" of feasible solutions situation. Our system guarantees that

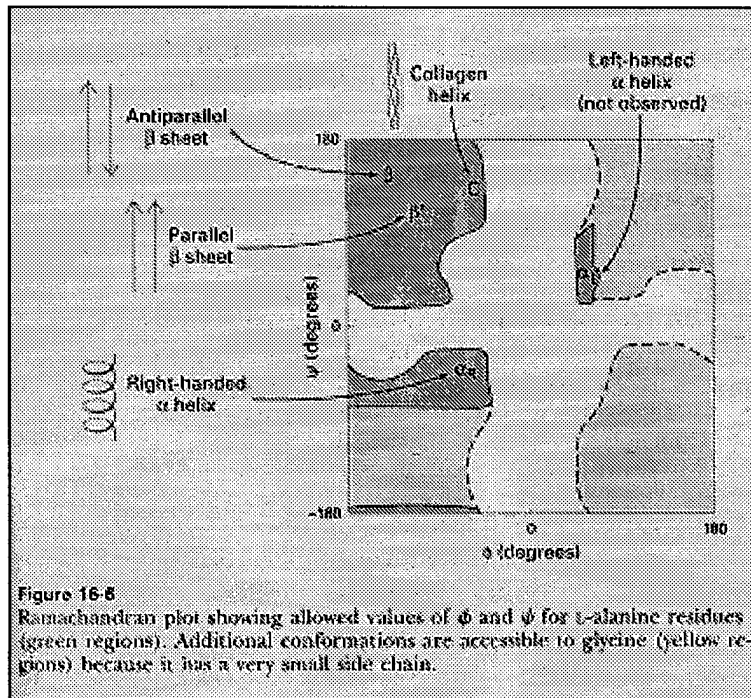


Figure 2.19. Stryers Ramachandran Plot [140]

the chromosome encoding mechanism ensures that the allowable genes are represented and maintained throughout all GA operations.

AFIT has devised another scheme to represent the search space using the Ramachandran Plots and affine transformations on the Φ - and Ψ -axis that ensures all GA operators, retain “feasible” solutions. The Ramachandran Plot is the key to AFIT’s system! At first glance, Figure 2.19 [140] makes it seem as if there are four distinct allowable regions based upon the values specified for (Φ, Ψ) . But after a simple coordinate transformation, it is easy to see that the Ramachandran Plot doesn’t actually create four regions, but rather just one smaller region within the complete space illustrated in Figure 2.21. The transformation is mathematically defined in Figure 2.20.

Still there are “infeasible” regions within 2.20. (i.e., The white space surrounding the darkened “bubble” represents unreachable Φ and Ψ angles.) Therefore, Deerman relocated the coordinate system origin, $(0,0)$, to correspond to two tangent lines and restricted the lengths of the axes to only span the feasible region - see Figure 2.21 and Figure 2.23.

$$\begin{aligned}
 &\text{If } \Phi_{new} \leq 180^\circ \text{ then } \Phi_{normal} = \Phi_{new} \\
 &\text{If } \Phi_{new} > 180^\circ \text{ then} \\
 &\quad \Phi_{normal} = |\Phi_{new} - 360^\circ| \\
 &\text{If } \Psi_{new} \leq 180^\circ \text{ then } \Psi_{normal} = \Psi_{new} \\
 &\text{If } \Psi_{new} > 180^\circ \text{ then} \\
 &\quad \Psi_{normal} = |\Psi_{new} - 360^\circ|
 \end{aligned}$$

Figure 2.20. First Transformation on Ramachandran Plot Equation

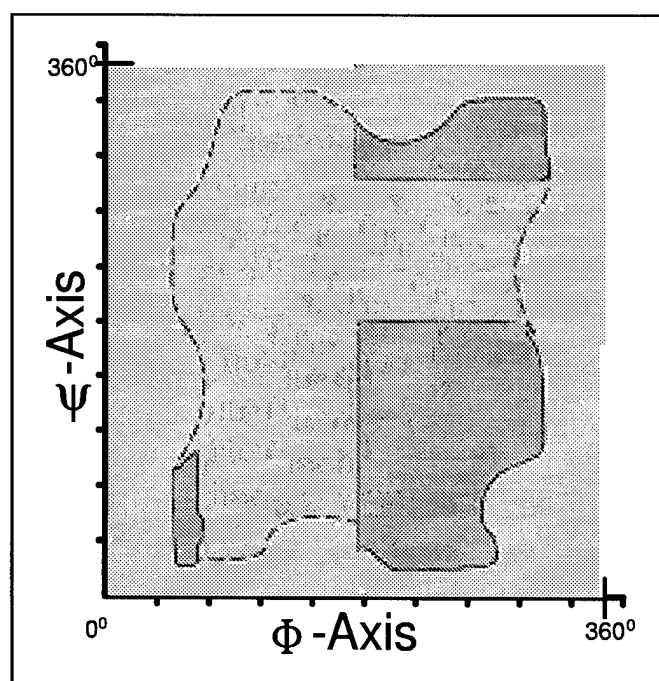


Figure 2.21. Applying First Transformation on Ramachandran Plot

$$\begin{aligned}\Phi_{new} &= \Phi' \left(\frac{(\Phi'_{upper} - \Phi'_{lower})}{360^{\circ}} \right) + \Phi'_{upper} \\ \Psi_{new} &= \Psi' \left(\frac{(\Psi'_{upper} - \Psi'_{lower})}{360^{\circ}} \right) + \Psi'_{upper}\end{aligned}$$

Figure 2.22. Second Transformation Equation

Now, the new Φ -axis (called Φ' -axis) corresponds to a tangent which intersects the point in the feasible region closest to the original Φ -axis, and likewise for the Ψ -axis (called Ψ' -axis). In past implementations, we have always assumed that the chromosomal encoding beginning at the origin (0°) and proceeding to 360° , represented by 20 to 210, respectively. But with this new coordinate system, this is no longer the case! The new (0,0) coordinate at the Φ' -axis Ψ' -axis intersection is approximately 20° up the Ψ -axis and 40° down the Φ -axis. Furthermore, the upper bounds of the Φ' -axis and Ψ' -axis are less by approximately 36° and 50° , respectively. This second transformation is mathematically defined by Figure 2.22.

Finally, (Figure 2.24) illustrates the 2nd transformation for the Φ -axis to the Φ' -axis.

Our scheme allows the GA to process the chromosome's backbone independent of representation and guarantees that for all GA operators the transformed chromosome is in the defined "feasible" region of the Ramachandran Plot. These transformations only work for the polypeptide backbone configuration and does not account for side chains! For the side chains, we consulted with Dr. Ruth Pachter (AFRL) to determine feasible ranges. Dr. Pachter validated the χ angles, as well as the backbone angles, proposed by Kaiser. [69] Therefore, the ranges he proposed for the backbone angles and side chains are used as our limits as well. This allows us to make direct comparison between his work and ours.

Our constraint system incorporates the chromosome encoding explicitly (see Table 2.6 and Table 2.7 for our limits [41, 69]), and the transformations into the proper angular configurations are accomplished in the objective function (e.g., within AFIT's CHARMm energy model implementation). This ensures that all future AFIT PSP researchers can

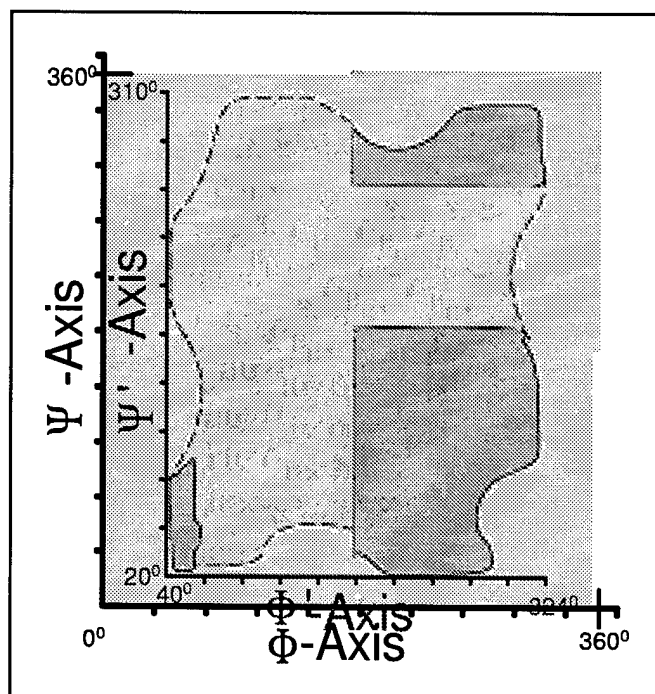


Figure 2.23. Applying Second Transformation on Ramachandran Plot

Table 2.6. Loose Constraints for Met-Enkephalin

Dihedral	Midpoint	Radius	θ_{\min}	θ_{\max}
$\Phi_{\text{non-glycine}}$	-120	90	-210	-30
Φ_{glycine}	-180	135	-315	-45
Ψ	60	150	-90	210
ω	-180	20	-200	-160
χ_1	-60 60 180	30	-75 45 -185	-45 75 -165

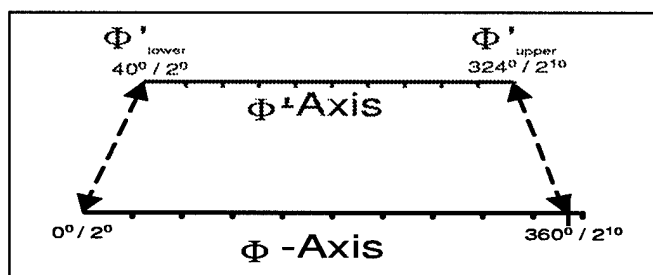


Figure 2.24. Φ Axis Transformation

Table 2.7. Loose Constraints for Polyalanine

Dihedral	Midpoint	Radius	θ_{min}	θ_{max}
ϕ	67.5	22.5	-90	45
ψ	30	30	-60	0
ω	180	20	-200	-160
χ_1	-60 60 180	30	-65 55 -185	-30 90 -150

use any contribution from this constraint system without having to manipulate their particular GA of choice. The appendices contain additional information concerning Newman projections illustrating the constraints in Table 2.6 and Table 2.7.

2.3 Landscape Visualization

The basic premise of scientific visualization is the use of computer-generated pictures to gain insight from the data. [109] This is still a very active and vital arena of research. While recent efforts in visualizing techniques have made some headway, it is unfortunate that the GA community still does not have a solid foundation of visualization techniques at their disposal. Of course, there are a number of commercial packages for visualizing proteins, polypeptides, and other molecule (i.e., Quanta, RASMOL, Cerius, VMD, etc.); however, most of these are only able to view the molecules postpartum. Currently, there is no software package available which allows the user to manipulate the molecule as it folds. This is due to the fact that to view the folding protein, the update rate of the visualization would need to be on the order of a femtosecond (e.g., 10-15 seconds). On the other hand, previous AFIT researchers have contributed to the expression of a particular protein's search space traversed by the GA. [25] As discussed earlier, the search space is "huge" when visualized in 2 dimensions but when viewed in its true n-dimensional form where n equals the number of independent variables entered into the energy function, the search

space is drastically reduced being bound in all dimensions by 360° ; however, a new problem arises. How do you visually represent a 25 dimensional picture, which is what is required for the small protein [Met]-Enkephalin! According to Deerman [25], some research indicates that the resulting image is a 25 dimensional funnel and his efforts to render this multi-dimensional landscape into just 3-dimensions resulted in the numerous data representation figures found in his thesis. For additional information with respect to Data Visualization the reader is encouraged to review Appendix H for additional information.

2.4 Deterministic Search Algorithms

In this section, we briefly look at how we could search the solution space via some deterministic algorithm. Recall from previous sections the discussion of the PSP problem domain puts the overall search space, for the [Met]-Enkephalin) protein, at 24^{1024} which is astronomical and the object, with a deterministic approach, would have to do a lot of pruning, if one expects to find the “global” minima energy value for this protein. In general, if the search space is not bounded, the size of the space to be searched is n^{1024} (with 10-bit representation for each dihedral angle), where n is the number of dihedral in the protein. As in all searches of this type, the objective is to provide criteria that allow us to prune the search tree as early as possible so we don’t have to explicitly search the entire search space. At best, we’d hope to find a “semi” optimal solution, as even with pruning, the landscape’s complexity is such that the algorithm probably searches a good portion of the landscape.

Perhaps the easiest to implement and to conceptualize would be a depth-first search approach. A depth-first search algorithm could be initialized with all dihedral angles at one extreme or the other and then by incrementing certain dihedral angular values, explore down a specific branch. Another method one might employ is a breadth-first search where all branches of the search tree are looked at level by level. Finally, a combination of the two previous methods could be employed. Regardless of the deterministic exploration method, the crux of the problem is its search space size. In order to make the problem even somewhat manageable a very very good pruning mechanism would have to be employed. This however, is the crux of the problem. Figuring out how best to prune is not an easy task

and is not explored further. Suffice it to say the difficulties in conducting a deterministic search on an NP-complete problem is doable; however, one would have to wait around for a very long time for any problem of sufficient size. To finish in any reasonable time, a pruning mechanism that prunes 99.99% of all branches is required—especially as the size of the problem is increased.

2.5 Stochastic Search Algorithms

Stochastic-based searching is nothing more than a probabilistic search. The range of stochastic algorithms start from exceedingly easy (random search) to the Simple Genetic Algorithm, Selfish Gene, etc. to more complex algorithms such as the fast messy Genetic Algorithm (fmGA). Essentially all stochastic search algorithms are based on a probabilistic engine that contains some uncertainty associated with a step within the search process. For instance, the uncertainty of a *purely* random search algorithm is in how the next member is selected—through a process of randomly generating the next individual to evaluate. In the case of something more complex, such as the fmGA, there are a number of probabilistic elements used in the algorithm as a decision basis of what to do next. The key to improving the algorithm is to incorporate generic concepts that are not related in anyway to a specific problem domain. For instance the fmGA takes advantage of the building block concept [46] and attempts to stochastically create a set of good building blocks in which to operate upon.

2.5.1 Genetic Algorithms (GAs). Genetic algorithms (GAs) are a stochastic search/optimization technique loosely based on natural evolution and the Darwinian concept of “Survival of the Fittest” [46, 63]. A generalized genetic algorithm consists of a *population* of *encoded* solutions that are manipulated by a set of *operators* and evaluated by some *fitness function* that determines which solutions survive into the next *generation*.

For our purposes, search and optimization techniques fall into the two broad categories previously mentioned : deterministic (a combination of calculus-based and enumerative) and stochastic (random) methods [46:2]. Greedy algorithms, calculus-based methods, and tree/graph search techniques are all examples of deterministic approaches [3]. These

methods have been successfully used to solve a wide variety of problems. However, there is an even greater number of problems where deterministic methods fail miserably [46:3–6][39][35]. The main stumbling block for deterministic methods is their requirement for some amount of problem specific knowledge to direct or limit the search. For example:

1. Greedy algorithms assume optimal sub-solutions are *always* part of the optimal solution [3, 76]
2. Calculus-based methods require continuity [80]
3. Tree/graph search techniques need problem specific heuristics/decision algorithms to limit the search space [39, 115]

A partial list of problem characteristics that make deterministic search techniques unsuitable for a particular problem includes: multi-modal and/or discontinuous solution spaces, exponential search spaces (NP-complete problems), and limited domain knowledge (no heuristics). Problems that exhibit one or more of these characteristics are called *irregular* [79].

Stuart Kauffman's tunable NK model of fitness landscapes [72] is a random energy model, similar to a spin glass [30], that is designed to capture the statistical structure of the rugged multi-peaked fitness landscapes that we see in nature. In the NK model, N represents the number of genes in a haploid (singular) chromosome and K represents the number of linkages each gene has to other genes in the same chromosome. Kauffman's motivation in creating the NK model was the study of co-evolution and the conditions under which Nash equilibria is attained. Nash equilibria are states in which all interacting components are optimal with respect to each other [105].

Several observations can be made from Kauffman's studies of the NK model. As K increases, the number of peaks in the fitness landscape increases and the landscape becomes more rugged. By rugged we mean that there is a low correlation between the fitness and similarity of genotypes, where the similarity metric used is Hamming distance. The value of K ranges from 0 to $N-1$. The extreme case of $K=0$ produces a highly correlated landscape with a single peak, while the other extreme of $K=N-1$ produces a landscape that is completely uncorrelated and has very many peaks. Another interesting observation is that as both N and K increase, the height of an increasing number of fitness peaks falls towards the mean fitness. This phenomenon, which Kauffman refers to as a "complexity

catastrophe", is a result of conflicting constraints among the genes. From a function optimization perspective, searching for peaks with high fitness in this case is analogous to looking for a needle in a hay stack.

The Kauffman's NK model shows that, due to its high level of epistatic interactions, the PSP is extremely multi-modal. Also, his model shows that as the complexity increases, basins of attractions rise toward the mean fitness. That is, not only are there many local minima, there is very little difference between local optima and the global optimal solution [73]. Likewise, Ngo, Marks, and Karplus provide a convincing argument that the PSP is NP-complete [108].

Stochastic search approaches (simulated annealing, evolutionary strategies, evolutionary programming, genetic algorithms, monte-carlo techniques) were developed to supplement deterministic techniques [99, 46]. The single requirement for stochastic methods is a function that assigns fitness values to possible solutions. Although these methods cannot guarantee the optimum solution, in general, they can provide *good* solutions to a wide range of problems that may be irregular and/or exponentially too large for deterministic methods [46, 76].

2.5.2 The Schema Theorem and Deception. While the SGA is somewhat good at finding solutions, there are inherent problems depending on the class of problems being solved. The following discussion on Schema theory and the idea of deceptive functions (which SGA's have difficulty in solving) provide additional insight into the development of current GAs.

Schemata are templates that define sets of strings with the same values at certain string positions and are represented using an additional *don't care* symbol (*) [46:19,29]. For example, the schema *101 represents the set of strings {0101, 1101} and the schema 1*0*01 defines the set {100001, 100101, 110001, 110101}. The *defining length* ($\delta(H)$) and *order* ($o(H)$) are two values associated with a particular schema H . The defining length of a schema is a measure of the distance between the first and last fixed positions. The order

of a schema is the number of positions with fixed values. Using the sample schemata from above $\delta(*101) = 4 - 2 = 2$, $o(*101) = 3$, $\delta(1*0*01) = 6 - 1 = 5$, and $o(1*0*01) = 4$.

$$m(H, t + 1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m \right], \quad (2.1)$$

The Schema Theorem, represented by equation 2.1, establishes a lower bound on the number of representatives schema H will have in the next generation ($m(H, t+1)$) based on the:

1. number of representatives in the current generation ($m(H, t)$),
2. fitness of schema H vs the population average fitness ($\frac{f(H)}{\bar{f}}$),
3. string length, defining length, and probability that schema H is destroyed by crossover ($p_c \frac{\delta(H)}{l-1}$), and
4. order and probability that schema H is destroyed by mutation ($o(H)p_m$).

Although it would appear that genetic algorithms operate only on the specific strings in a population, it has been shown that many of the 2^l schemata in each string are processed simultaneously (*implicit parallelism* [62:71–72][46:40]). The Fundamental Theorem of Genetic Algorithms (Schema Theorem) states that all schemata receives representation in the next generation proportional to the ratio of their fitness to the average fitness of the population. This representation is reduced by the amount of disruption that crossover and mutation can cause to a schema. More succinctly,

short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations [46:33].

2.5.3 Messy Genetic Algorithm (mGA). Genetic algorithms are designed to take advantage of the *building block* theory [62, 46, 38]. The main idea is that small pieces of a solution which exhibit above average performance are combined to create larger pieces of above average quality, which are themselves recombined into larger pieces, and so forth.

Simple genetic algorithms suffer from the fact that the “pieces” that form the building blocks must be put next to each other explicitly in the fixed encoding or else they are more likely to be disrupted by crossover. This problem is magnified when competing schemata (schemata with different values at similar defining positions) define locally optimal solutions. *Deception* occurs when locally optimal building blocks are selected instead

of globally optimal ones. Messy genetic algorithms (mGAs) were designed to deal with these problems by encoding the string position (locus) along with its value (allele). This gives a messy genetic algorithm the ability to search for the “true” building blocks of the problem and create tighter *linkage* for those genes than a fixed position encoding would allow [51]. The mGA encoding scheme also allows *under-specified* and *over-specified* strings to exist in the population. Under-specified strings don’t have an allele defined for every locus and are evaluated with the aid of a locally optimal *competitive template* that supplies values for the unspecified genes. Over-specified strings contain multiple alleles specified at the same locus and are processed in a left-to-right fashion which sets the gene to the value encountered first. The desire to create and manipulate superior building blocks is the motivation behind messy genetic algorithms [51, 49, 50].

2.5.3.1 Messy Genetic Algorithm Operators. Messy GAs use variations of the same genetic operators used by simple GAs. In the few implementations of mGAs that exist [50, 51, 49, 88], tournament selection has been used instead of proportional or rank-based selection because of its desirable performance characteristics [50:50][47, 33]. The tournament selection operator also has a *thresholding* mechanism added to it which ensures that strings have a number of positions in common before competition is allowed [49:424–427]. Crossover is replaced by a combined *cut-and-splice* operator that works on variable length strings. As the names suggest, *cut* divides a string into two smaller pieces and *splice* concatenates two strings to form a single, longer string. A mutation operator that can change a gene’s value or its position has been described, but it is not used in any mGA implementations [51:504].

Messy GAs employ a different initialization strategy compared to SGAs. The main processing loop of an mGA is composed of *primordial* and *juxtapositional* phases. During *partially enumerative initialization (PEI)*, exactly one copy of each possible building block of the specified size (k) is generated. Thus, the initial population size for a messy GA is generally quite large ($2^k \binom{l}{k}$) [51:420]. The primordial phase serves two basic purposes: enrich the population with above average building blocks and reduce the population to a size that can be efficiently and effectively processed by the juxtapositional phase. Tourna-

1. perform partially enumerative initialization
evaluate fitness of all population members
2. for $i = 1$ to the maximum number of primordial generations
 perform tournament selection
 if (a suitable number of generations have transpired) then
 reduce the population size
 end if
end loop
3. for $i = 1$ to the maximum number of juxtapositional generations
 perform cut-and-splice
 perform other operators (currently not used)
 evaluate fitness of all population members
 perform tournament selection
end loop

Figure 2.25. Pseudo Algorithm for Messy GAs

ment selection, the only active operator during the primordial phase, fills the population with above average building blocks, then periodically the population size is halved. No additional fitness evaluations are required during the primordial phase. The juxtapositional phase is most similar to the main processing loop of a simple GA [51:506]. Cut-and-splice and many other genetic operators are applied, fitness evaluations are performed on the newly created strings, and tournament selection bolsters the next generation with highly fit solutions. A pseudo algorithm for messy GAs is shown in Figure 2.25.

2.5.3.2 Messy Genetic Algorithm Parameters. The major parameter settings associated with messy GAs are population size, cut-and-splice probabilities, and a schedule for reducing the population size. Initial population size can be calculated once the string length and block size have been determined. String length is simply a function of the encoding used, but block size is a problem dependent quantity that may be difficult to estimate. The final population size at the end of the primordial phase is even less quantifiable! The splice probability is consistently set to 1.0 with the following rationale: the primordial phase ends with a population of optimal building blocks which should only require assembly to form a complete string that is a near-optimal solution [50:25]. The chosen cut probability is scaled by the current length of a string so that longer strings are more likely to be cut than shorter strings. The schedule for reducing population size

during the primordial phase typically allows for two or three generations of enrichment followed by cutting the population in half [51:505]. No theoretical or empirical work has been accomplished to provide any guidance for final primordial population size, cut probability, or population reduction schedules for messy GAs.

2.5.3.3 Why Does the Messy GA Work? The Schema Theorem (Equation 2.1) is directly applicable to messy genetic algorithms. The rationale for messy genetic algorithms follows from the theorem's interpretation: "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations." If the building blocks of a problem aren't encoded as short, low-order schemata, then crossover and mutation disrupts the formation of those building blocks.

For problems using a fixed encoding, where the identification of building blocks is prohibitive or impossible, Goldberg has calculated the *normalized expected defining length* ($\frac{\langle \delta \rangle}{l+1}$) for k -sized building blocks (Equation 2.2). The normalized expected defining length is a measure of the mean length of the schemata that make up the building blocks of a randomly encoded problem. The interpretation is that an arbitrary encoding is highly unlikely to establish tight linkage for the building blocks of a problem [51:498-499].

$$\frac{\langle \delta \rangle}{l+1} = \frac{k-1}{k+1} \quad (2.2)$$

Messy genetic algorithms take advantage of the Schema Theorem by searching for both the defining positions and gene values of the building blocks using PEI and the primordial phase. Then the juxtapositional phase of the messy GA starts with "short, low-order, above-average" schemata that are also "short, low-order, above-average" building blocks!

2.5.3.4 Complexity Analysis. Because of the partially enumerative initialization (PEI), the time complexity of messy GAs is $O(l^k)$. This compares unfavorably with the rest of the algorithm which is only $O(l \log l)$ [49:420-422]. Space complexity remains unchanged from simple genetic algorithms. However, the constant term is generally larger and the population size (n) is *much* larger! As is the case with simple genetic al-

gorithms, the time complexity of the evaluation function usually dominates that of the control sequence.

2.5.4 Fast Messy Genetic Algorithm (fmGA). The advantage messy GAs have over simple GAs is the ability to create tightly linked building blocks for the optimization of deceptive problems. The disadvantage associated with this better processing is the time complexity of the initialization phase which dominates the mGA algorithm [49:422]. Fast messy GAs are a messy GA variant designed to reduce the complexity of the initialization phase, and thus the overall algorithm time and space complexity [48:59].

2.5.4.1 Fast Messy Genetic Algorithm Operators. PEI and the selection-only primordial phase of mGAs are replaced by *probabilistically complete initialization* (PCI) and a primordial phase consisting of selection and building block filtering (BBF) in fmGAs. PCI and BBF are an alternate means of providing the juxtapositional phase with highly fit building blocks [48:59–61].

PCI is used to create an initial population whose size is equivalent to the population size at the end of the primordial phase of mGAs. The length of these strings is typically set to $l - k$. The primordial phase then alternately performs several tournament selection generations to build up copies of highly fit strings followed by BBF to reduce the string length toward the building block size (k). Building block filtering is a simple process that randomly deletes several genes from a string. The juxtapositional phase is the same as in mGAs. A pseudo algorithm for fast messy GAs is shown in Figure 2.26.

2.5.4.2 Fast Messy Genetic Algorithm Parameters. Fast messy GAs need a building block filtering and thresholding schedule instead of the population size reduction schedule required by mGAs. Goldberg provides formulas for deriving schedules [48:60–61], but the formulas contain additional parameters and no guidance is given for choosing their values. The remainder of mGA parameters are used by fmGAs as well.

2.5.4.3 Why Does the Fast Messy GA Work? Fast messy GAs are governed by the Schema Theorem (Equation 2.1) just like mGAs. The difference relates to how the

1. perform probabilistically complete initialization
evaluate fitness of all population members
2. for $i = 1$ to the maximum number of primordial generations
 perform tournament selection
 if (a building block filtering event is scheduled) then
 perform building block filtering
 evaluate fitness of all population members
 end if
end loop
3. for $i = 1$ to the maximum number of juxtapositional generations
 perform cut-and-splice
 perform other operators (currently not used)
 evaluate fitness of all population members
 perform tournament selection
end loop

Figure 2.26. Pseudo Algorithm for Fast Messy GAs

population of “good” building blocks is created for processing by the juxtapositional phase. Goldberg performs a detailed analysis to show that a much smaller initial population of long strings (PCI) can be manipulated (through BBF) to create a population of “good” building blocks just as effectively as PEI and the primordial phase of mGAs [48:60–61].

2.5.4.4 Complexity Analysis. Reducing the overall time complexity of the algorithm is the main reason for switching from mGAs to fmGAs. PCI and BBF result in a time complexity of $\mathcal{O}(l \log l)$ for initialization and the primordial phase combined [48:61]. Thus, the design goal has been met—fmGAs exhibit better efficiency than mGAs ($\mathcal{O}(l \log l)$ vs $\mathcal{O}(l^k)$) and preserve their effectiveness. Space complexity for fmGAs remains unchanged from SGA’s and mGA’s ($\mathcal{O}(nl)$) and populations can be sized much smaller than mGAs. Again, the time and space complexities of the evaluation function usually dominate those of the control sequence.

2.6 Parallel Architectures and Genetic Algorithms

2.6.1 Homogeneous vs. Heterogeneous Computing Environments. There are a number of differing parallel architectures available to researchers requiring *high* computational processing powers. While there are a number of specifications, with respect to the

hardware implementation of a parallel computer system, the most fundamental specification is that of the “processor” speed. If all processors for a single parallel architecture are identical, then that parallel architecture is considered homogeneous. If, however, there are varying processor speeds, then the system is considered heterogeneous.

Homogeneous parallel systems were the first to arrive on the computer market in the form of mini and supercomputers. [146] With the advent of the personal computer and its subsequent rise in computational power and concurrent reduction in price, many scientific communities opted to take advantage of the processing capability per unit of cost of the personal computer over that of the supercomputer. While one could still build a homogeneous parallel system with personal computers, more often than not, the parallel system is changed into a heterogeneous one due to the fact that the computer manufacturers continually inject faster/better machines at competitive costs and it is cheaper replacing a limited number of machines annually than replacing them all every other year.

2.6.2 Parallel Genetic Algorithms. There are two major concerns when parallelizing any algorithm: is the parallel algorithm correct and is it faster than the serial version? Correctness is an issue because we have even greater difficulty verifying parallel algorithms than we have for sequential programs [83]. Given that the parallel algorithm is correct, speedup is the primary goal of parallelization [12]. A tradeoff analysis is generally required to determine if the estimated benefits warrant the expenditure of resources to parallelize an algorithm. There is evidence to suggest that parallelizing genetic algorithms is worthwhile and should be examined further [27, 141, 117, 134, 53].

2.6.3 Parallel Decomposition Techniques. Data and control decomposition are alternate means of dividing a problem into portions that can be worked on simultaneously. In general, data decomposed algorithms perform the same operations of subsets of the input (*data parallelism*) and control decomposed algorithms that perform different operations on the total input [84]. In either case, the results are recombined in some fashion to obtain the final result(s). Genetic algorithms are easily parallelized because they are highly data decomposable (although control decomposition is not impossible, especially as the complexity of operators and evaluation functions grow). Parallelizing GAs using

data decomposition can be as simple as running multiple copies of the same program on different processors, each starting with a different random number seed, and then choosing the best result from all runs. Data parallelization techniques are also amenable to static load balancing because their computation and communication patterns are regular [31, 84]. This does not imply that all processors are searching in equally promising portions of the search space. Thus, some efficiency may be lost to sub-populations that are searching similar solution neighborhoods or stuck in local optima. Two models, which lie at opposite ends of a granularity spectrum, have been proposed for parallel genetic algorithms—the *island model* (course-grained) and the *neighborhood model* (fine-grained) [26, 53]. These models are designed to improve the simplistic parallel approach by sharing near-optimal results with some portion of the global population.

2.6.4 Island and Neighborhood Model. The island model is an extension of the simplistic approach where the total population is divided into sub-populations which are distributed among the processors. The sub-populations evolve in parallel; however, at certain time intervals, a *migration* occurs where solutions are communicated between processors [26:10]. Migration rates, migration selection strategies, and migration patterns are additional parameters with associated design decisions that must be defined for the parallel genetic algorithm. Near-linear speedup is expected and has been observed for island model parallel genetic algorithms [4:60][141, 10]. The time complexity of island model GAs is $\mathcal{O}(\frac{nl}{p})$, where p is the number of processors, n is the population size, and $p \ll n$ [26]. As $p \rightarrow n$, the resulting small subpopulation size increases the ratio of communication time to compute time and the speedup becomes much less than linear. Island model GAs are typically used on course-grained or multiple-instruction-multiple-data (MIMD) architectures [76].

The neighborhood model splits the population up spatially in a two- or three-dimensional grid. This grid and the definition of a neighborhood limits the interaction of individuals in the population. Typically, a single string is assigned to each processor, therefore crossover and selection must be modified because their operation is distributed across more than one processor. Although their convergence characteristics have been

observed to be better than the Island Model [26:24], neighborhood model parallel genetic algorithms don't exhibit speedup because they assume $n = p$. Therefore no speedup can be obtained because more/fewer processors are not allowed. The neighborhood model is most often compared to the simple GA for time complexity analysis ($\mathcal{O}(s + l)$ vs $\mathcal{O}(nl)$ where s is the neighborhood size) [26:24]. Neighborhood model GAs are generally implemented on fine-grained or single-instruction-multiple-data (SIMD) architectures [26].

2.6.5 Typical Metrics to Compare Parallel Systems. There are a number of metrics used to compare parallel computing systems. Many of the metrics deal with specific aspects of the parallel system. For instance there are metrics that evaluate the static interconnection of networks such as [78]:

1. Diameter - the maximum distance between any two processors in the network
2. Connectivity - a measure of the multiplicity of paths between any two processors
3. Bisection Width - the minimum number of communication links that have to be removed to partition the network into two equal halves
4. Bisection Bandwidth - the minimum volume of communication allowed between any two halves of the network with an equal number of processors
5. Channel Rate - the peak rate at which a single physical wire can deliver bits
6. Channel Width - the number of bits that can be communicated simultaneously over a link connecting two processors
7. Cost - there are a number of ways to calculate this; however, one way of defining the cost of a network is in terms of the number of communication links or the number of wires required by the network

These types of metrics are used to evaluate the actual parallel system. There are other metrics associated with measuring the performance of parallel algorithms run on parallel systems. A discussion of the metrics used in this thesis can be found in Chapters 4 and 5.

2.7 Random Number Generators (RNG)

The stochastic nature of a GA is totally dependent upon the implemented RNG. Dymek's Appendix A [28] and [110] covers the importance of random number generators due to this heavy reliance. The random number seed dictates where in the problem's search space the GA begins searching and how various segments of the code are implemented. Therefore, it is extremely important that "good" random number generators are used. A good random number generator is defined as one in which no "perfect correlation" occurs [28]; a perfect correlation between two RNGs results in the same instantiated behavior from two separate GA executions. At first glance, this sounds much worse than the situation warrants. For the purposes of validation of experiments, two separate GA test runs, which start with the same random number seed, should result in precisely the same GA behavior. On the other hand, RNGs are only pseudo-random. Appendix I contains additional details about RNGs and the reader is encouraged to read the appendix to gain a better insight into RNGs and concerns and issues associated with utilizing them in both serial and parallel platforms.

With respect to this work, the fmGA algorithm is applying a masking function to generate random numbers and/or the built-in gnu C RNG function call. It is assumed that their (Gates and Merkle [41, 93]) prior work was correct in its implementation of the RNG.

2.8 Summary

Genetic algorithms are semi-optimal search/optimization techniques capable of finding "good" solutions to problems that are intractable for deterministic methods. Many theories and conjectures have been proposed based on both mathematical analysis and toy problem experimentation with genetic algorithms, their control structure, and different genetic operators. Genetic algorithms have been applied to some relatively small real-world problems with good results.

Five iterations of theses at AFIT have already examined GAs applied to the tertiary structure prediction aspect of the PFP [4, 41, 43, 69, 25]. To some extent, their work has

focused on perfecting the GA as a robust problem solver independent of the application in the spirit of Holland and his students. This thesis, on the other hand, adopts the philosophy of Michalewicz in the title of his book on GAs, *Genetic Algorithms + Data Structures = Evolution Programs* [99]. That is, it actively pursues domain knowledge to guide the search process while maintaining the stochastic nature of GAs and attempts to build upon Gate's [41] previous work. The above section on the proteins has built the basis for this domain knowledge. Later chapters detail the specific implementation.

III. Methodology: Incorporation Secondary Structure into the fmGA

3.1 Introduction

As with any major undertaking, be it the construction of a new building, the development of a new drug, or even the building of an elaborate model airplane for one's boy, a clear concise method or set of instructions is *always* beneficial. Our efforts to incorporate germane secondary structure information into the fmGA is no different and this chapter provides those details needed to accomplish our goal. The chapter is organized by first providing some background information, then a discussion about the high-level and low-level designs, followed by a discussion of the energy fitness function, and lastly a closer look at the actual modifications required to incorporate secondary structure information (major enhancements) as well as other (minor) enhancements to improve both the efficiency and effectiveness of the fmGA algorithm.

3.2 Background

Previous research at AFIT resulted in a number of parallel and serial genetic algorithm implementations and evaluation functions for several domains [14, 15, 18, 37, 52, 66, 73]. Collectively, these are known as the AFIT Genetic Computation Toolkit (AGCT). The current state of the AGCT toolkit is in Figure 3.1. The contributions of this research to the toolkit are marked by (Michaud). The purpose of this chapter is to document the design decisions, implementation details, and interface requirements of the modified fmGA. Deerman's work [25] provided a sound description of the CHARMM fitness function and his efforts to incorporate the Ramachandran constraints and while Gates' thesis outlined an extensive UNITY representation of the fmGA [41]. This chapter concentrates on only the modifications to Gates and Merkle's parallelized fmGA. The reader is reminded that the intent of this thesis effort is to improve the overall performance by reducing computational requirements and improving the effectiveness of the fmGA through the incorporation of additional domain knowledge as well as modifying the way the fmGA currently runs.

3.2.1 AFIT's PSP fmGA Implementation. AFIT has more than 10 years of research effort with respect to finding a viable solution to the geometrical conformation of

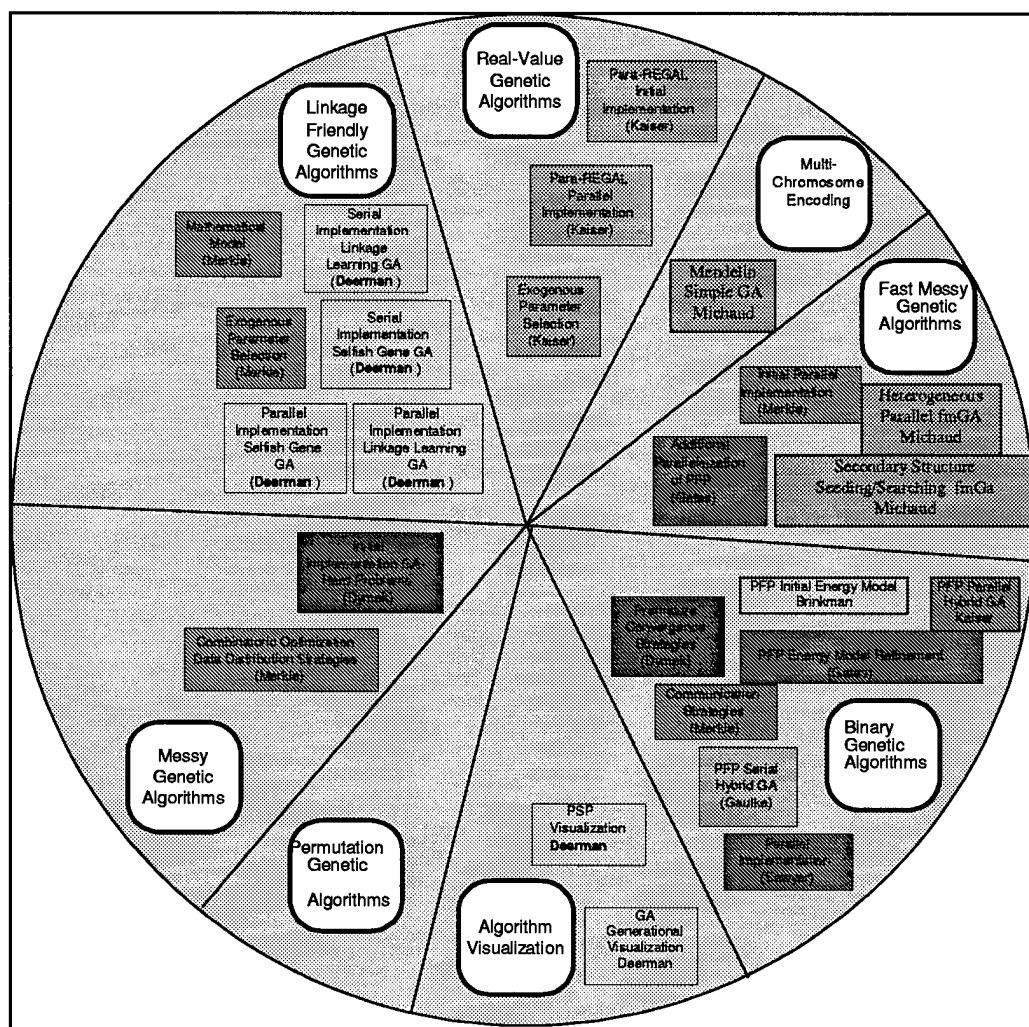


Figure 3.1. AFIT's ToolKit

the Met-Enkephalin protein. Previous master student's efforts have played an important part in our "collective" understanding of the Protein Structure Prediction problem, as well as our ability to employ genetic algorithms to finding a viable solution.

3.2.2 Evolution of AFIT's GAs. Many genetic algorithm designs and implementations exist. Holland provided a sound mathematical conjecture for families of GAs based on reproductive plans and three genetic operators (crossover, mutation, and inversion) [62] and later De Jong implemented a subset of one of those families based on classes of reproductive plans [22]. Typically, designs are functionally decomposed to facilitate the

Table 3.1. Available Genetic Algorithm Implementations

Implementation	Characteristics
Genesis [56]	binary alphabet, functionally decomposed good modularity, written in C command line interface, functional design
OOGA [20]	real-valued and combinatoric representations and operators written in Lisp, object-oriented design
Splicer (NASA)	binary and combinatoric representations and operators written in C, X-Windows interface, functional design

construction of GA workbenches. These workbenches are used to study the behavior of many GA operators. Object-oriented designs also exist, and lend themselves to the examination of representation issues and production systems. Table 3.1 (taken from Gate's thesis [41]) lists a few of the implementations that are widely used along with some of their major characteristics.

AFIT's work with GAs is either directly or indirectly based on the GENESIS software package. According to Gates, the main reasons Genesis was used and continues to be used as a general platform for GAs at AFIT are:

1. GA code must be easily portable to multiple serial and parallel hardware platforms.
2. The GA must contain an I/O interface that provides simple parameter input and meaningful progress and convergence output.
3. Our requirements are for a GA workbench to examine the effects of operators and control structures on function optimization and combinatoric search.
4. Our access to Genesis predates access to all other software packages.

According to Gates, item 1 is the most restrictive as it limits our choices to C and FORTRAN implementations. While the original analysis and selection motives have been lost in revisions of the Compendium of Parallel Programs for the Intel iPSC Computers [79], Gates hypothesized that Item 4 was a major reason for picking and sticking with Genesis.

3.2.3 Inputs into AFIT's PSP fmGA. There are a number of input files that are required to properly run the PSP fmGA. This was true before any modifications this effort has caused and is still true. In general there are input files that deal directly with the defining of a Protein (MOL.RTF, PARM.PRM, and mol.z) and a single file that is used to initialize fmGA parameter settings. Brinkman's thesis [4] provides a detail description

of the protein defining input files. The original input file used by Gates [41] (see Table 3.2) was slightly modified by Merkle in an attempt to analyze parallel performance of the parallel fmGA (pfmGA) [90] (see Table 3.3).

The fmGA requires a number of parameters that go beyond the normal GA's mutation rate, crossover rate, in fact, the fmGA requires parameters to create building blocks in the primordial phase.

The parameters required for the Primordial Phase or Building Block Filtering stage are:

1. *Primordial Generations*—indicates the number of generations to use to generate the building blocks
2. *Building Block Size*—sets the final length of the building blocks to be created
3. *Scheduling Table (Cut_gen, Str_len, Threshold)*—this table dictates the rate at which a fully specified string (initial state) is reduced to the building block size specified. The cut_gen value identifies the generation in which random bits are selected and tossed out, str_len indicates the length of the string after randomly deleting bits, and threshold is used during the none bit-reducing generations of the Primordial phase as a measure of likeness (hamming distance if you will) and is used during selection (i.e. if two members chosen at random have a threshold of bits positions—loci—that are defined then they can be compared)
4. *Shuffle Number*—is essentially a range of population members that can be looked at during the selection process. For instance if the shuffle number was 2 then only the following two members of the population would be looked at, assuming the first and second population member failed to be within a specified threshold (hamming distance) from the first population member.

The parameters used in the Juxtapositional phase are:

1. *Overflow*—used during the cut and splice subfunction to indicate the maximum length allowed during the recombination of building blocks
2. *Total Generations*—used as a stopping criteria for the Juxtapositional Phase

Table 3.2. Gate's Original Input File for the Serial Version of the fmGA

```

Random Seed = 1234567890
Experiments = 10
String Length = 240
Block Size (1 - String\ Length) = 5
Genic Alphabet = 01
Shuffle Number (>1) = 0
Cut Probability = 0.02
Splice Probability = 1.0
Primordial_Generations = 200 200 200 200 200
Total_Generations = 400 400 400 400 400
Overflow (>1.0) = 1.6
n_a = 200 200 200 200 200
Cut_gen Str_len Threshold
0 216 194
7 185 143
11 157 107
15 135 84
19 115 64
23 98 47
29 84 38
35 72 31
41 61 25
47 53 21
53 45 17
59 39 15
65 33 12
71 29 10
77 24 8
82 21 7
87 18 6
92 15 5
97 13 4
102 11 4
107 9 3
112 8 3
117 7 2
122 6 2
127 5 3

```

Table 3.3. Merkle's Modified Input File for the Parallelized Version of the fmGA

```

Random Seed = 987654321
Experiments = 10
String Length = 240
Block Size (Min and Max) = 6 10
Genic Alphabet = 01
Shuffle Number (>1) = 2
Cut Probability = 0.02
Splice Probability = 1.0
Overflow (>1.0) = 1.6
Population Recombination = I
Initialization Flag = C
Distinct Competitive Templates = Y
Competitive Template Supplied = N
Global Selection = N
Migration = N
Minimization Probability = 0.02
Replacement Probability = 0.01
Primordial_Generations = 200 200 200 200 200
Total_Generations = 400 400 400 400 400
n_a = 800 800 800 800 800

Cut_gen  Str_len  Threshold
0         216     194
7         185     143
11        157     107
15         135     84
19         115     64
23          98     47
29          84     38
35          72     31
41          61     25
47          53     21
53          45     17
59          39     15
65          33     12
71          29     10
77          24      8
82          21      7
87          18      6
92          15      5
97          13      4
102         11      4
107          9      3
112          8      3
117          7      2
122          6      2
127          5      3

```

Other miscellaneous input parameters worth mentioning are:

1. *n_a*—used to define the size of the population set
2. *string_length*—directly related to the number of dihedral angles in the protein under-study and since each angle is represented by 10 bits then the Met-Enkephalin would have 240 for this value as it has 24 dihedral angles

During the parallelization of the fmGA, Merkle and Gates added the following parameters to allow for various parallel computational models [89, 90] (See Figure 3.3).

1. Population Recombination
2. Initialization Flag
3. Distinct Competitive Templates
4. Competitive Template Supplied
5. Global Selection
6. Migration
7. Minimization Probability
8. Replacement Probability

All of the following input parameters were added except Sweeps and Competitive Template Guesses which were moved from the "global.h" file to an input parameter (eliminated the need to recompile every time these two variables were changed).

1. Sweeps
2. Competitive Template Guesses
3. Secondary Structure Percentage
4. Good Population Percent
5. PopType 0-C 1-S 2-B 3-A
6. Initial Energy Cutoff Value
7. Protein Structure Used
8. Number of Residues in Protein
9. Heterogeneous

Each new parameter was required to allow the user to control how the operation of the algorithm performed for various additional features which were added in the code and are explained in the following sections of this chapter. Table 3.4 is the current input parameter file required to run the fmGA in an attempt to locate a reasonably good solution.

Table 3.4. Secondary Structure and Additional Parallel Input Parameters for the fmGA

```

Random Seed = 987654321
Experiments = 1
String Length = 560
Block Size (Min and Max) = 16 20
Genic Alphabet = 01
Shuffle Number (>1) = 2
Cut Probability = 0.02
Splice Probability = 1.0
Overflow (>1.0) = 1.6
Sweeps = 3
Competitive Template Guesses = 5
Secondary Structure Percentage = .00
Good Population Percent = .50
PopType 0-C 1-S 2-B 3-A = 2
Initial Energy Cutoff Value = 800.0
Protein Structured Used = POLY
Number of Residues in Protein = 14
Heterogeneous = N
Population Recombination = I
Initialization Flag = C
Distinct Competitive Templates = N
Competitive Template Supplied = N
Global Selection = N
Migration = N
Minimization Probability = 0.02
Replacement Probability = 0.01
Primordial_Generations = 200 200 200 200 200
Total_Generations = 400 400 400 400 400
n_a = 50 50 50 50 50
Cut_gen Str_len Threshold
0 540 500
7 480 460
11 440 420
15 400 380
19 380 360
23 360 360
29 340 320
35 300 280
41 260 240
47 220 200
53 180 140
59 148 128
65 125 105
71 109 99
77 94 83
82 81 67
87 68 56
92 55 45
97 43 34
102 30 24
107 20 14
112 19 13
117 18 12
122 17 11

```

3.3 *Modifications-Additions To AFIT's PSP fmGA*

This section describes in high-level psuedo-code the changes that were made to the fmGA in an attempt to improve overall efficiency and effectiveness. Subsections of this section address the following:

1. Porting the fmGA from NX to MPI
2. Fine Tuning the Local Search
3. Effects of the Initial Competitive Template
4. Heterogeneous Implementation
5. The Epoch fmGA
6. Searching for Secondary Structures
7. Population Seeding
8. Incorporating Ramachandran Constraints
9. Short-circuiting the Energy Fitness Function
10. Decimal Representation of Dihedral Angles as Output

3.3.1 Porting the fmGA from NX to MPI. While Merkle and Gates [89, 90] parallelized the fmGA for the Paragon Supercomputer using NX communication hooks and a separate version of the code using MPI communication hooks, no MPI version of their effort could be located. As a result, the parallel version of the code utilizing NX communications calls, as well as, references [93, 88] were used as a basis and aids, respectively, to port the NX-based parallel version over to MPI. This process though seemingly easy was much more difficult as there was no true direct mapping between the two and slight modifications in the data representation were required. However, these modifications were specific to communicating between the nodes and did not effect the fmGA and its implementation whatsoever. Since the parallelization and its effects on the fmGA were already addressed extensively in [42, 88, 79, 95, 96, 90, 93], additional information concerning the methodology employed is not be addressed. Appendix A discusses the porting of the fmGA from NX to MPI.

3.3.2 Fine Tuning the Local Search. The fmGA utilizes a local hill-climbing search method to derive the competitive template. There were a couple of issues that were addressed during this research. They are:

1. Should the sweeping method sweep from the left to right—in other terms, should we flip the least-significant or most-significant digits first (which corresponds to optimizing either the backbone or side-chain dihedral angles first)?
2. How do we ensure that we are not wasting computational power sweeping on a solution string that is already in a local minima?

The first item above hints at problem domain information because the domain information is inherently embedded within the solution string's structure and how that structure is mapped into the fitness function. With a randomly generated string, it is believed that the order of the sweeping method is not necessarily an issue with the exception that as you move from left to right, the most significant digits are changed and as a result the algorithm takes large steps during its hill-climbing search on a specific surface of the hyperplane that represents the entire solution space. It is not known if the left-to-right or right-to-left sweep has an advantage over the other with respect to a randomly generated string. Item 1 first became a problem when our research led us in the direction of seeding the population with a secondary structure embedded solution string that was subsequently hill-climbed. This is an issue because the ϕ , ψ , and ω backbone angles are considered to be within the constraint and the side chain ϕ , ψ , and ω angles were not and thus should be optimized around the ϕ , ψ , and ω backbone angles.

The second item identified was directly related to our effort of improving efficiency of the algorithm. Tests were accomplished and it was found that in general, sweeping through the solution string in an attempt to find an optimized solution, the algorithm will eventually not be able to find a better solution and thus waste computational power. It was felt that this should be remedied and thus a simple control loop was placed around the sweeping function that essentially would terminate the search if through one loop, no change had occurred.

```

Generate Competitive Template
Synchronous MPI Call to Find Best Competitive Template Across Machines
  For n = 1 to k
    Perform Probabilistically Complete Initialization
    Evaluate Each Population Member's Fitness (w.r.t. Template)
  // Building Block Filtering Phase
    For i = 1 to Maximum Number of Building Block Filtering Generations
      If (Building Block Filtering Required Based Off of Input Schedule)
        Then Perform Building Block Filtering
      Else
        Perform Tournament Thresholding Selection
      Endif
    End Loop (for i=1 to max bbf gens)
  // Juxtapositional Phase
    For i = 1 to Maximum Number of Juxtapositional Generations
      Cut-and-Splice
      Evaluate Each Population Member's Fitness (w.r.t. Template)
      Perform Tournament Thresholding Selection
    End Loop (for i = 1 to max number of Jux Gens)
    Synchronous MPI Call to Find Best Population Member Across Machines
    Assign Best Member as the Competitive Template
  End Loop (for n=1 to k)
Synchronous MPI Call to Find Overall Best Across Machines

```

Figure 3.2. pfmGA Algorithm Pseudocode

3.3.3 Heterogeneous Version of fmGA. Another issue that was of great concern was the waste of computational power in a heterogeneous environment. To enhance the algorithm, timing control mechanisms were incorporated into the algorithm that would essentially capture the processing power of all machines and inform each machine of the slowest one. Based on that information, each processor was allowed to do an extra amount of work based on its own speed and that of the slowest processor in the group.

In our work in [101], we fully test and report the results of this effort. Additional aspects not covered in the paper are discussed in Chapter IV. The high-level pfmGA algorithm is provided in Figure 3.2 and the changes required to optimize its operation in a heterogeneous environment are in Figure 3.3.3. The driving factor besides improving the effectiveness of the algorithm and improving overall utilization of the processing power of the cluster of machines used, was to derive a simple method in which one could dynami-

```

Initialize MPI
Start GCT Timer
Generate Competitive Template
Stop GCT Timer
If Extra Time is Available at this Juncture
    Generate Extra Competitive Templates
Endif
Synchronous MPI Call to Identify the Slowest Processor in Generating the CT
Synchronous MPI Call to Find Best Competitive Template Across Machines
For epoch = 1 to  $j$ 
    For  $n = 1$  to  $k$ 
        Perform Probabilistically Complete Initialization
        Building Block Filtering Phase
        Juxtapositional Phase
        If Extra time is available at this juncture
            Attempt to Optimize Population Members via the Sweep Function
        Endif
        Stop PhaseTimer
        Synchronous MPI Call to Identify the Slowest Processor to finish the three phases
        Synchronous MPI Call to Find Best Population Member Across Machines
        Assign Best Member as the Competitive Template
    End Loop (for  $n=1$  to  $k$ )
End Loop (for epoch= $1$  to  $j$ )
Synchronous MPI Call to Find Overall Best Across Machines

```

Figure 3.3. Heterogeneous pfmGA Algorithm Pseudocode

cally add or remove machines from a group without having to conduct a detailed analysis of the new machines. The algorithmic changes proposed comprise all of the potential variables one could look at in determining load or task balancing into a single parameter and that is “time”! If time is available, then do extra work. The extra work that the pfmGA can do depends on what point the algorithm it is at. For instance right after the “Generate Competitive Template” if time is available the algorithm is instructed to generate additional competitive templates based on specific input parameters (not changed) until it doesn’t have enough time available to conduct another. The other location in the algorithm where extra work can be done is at the end of the Juxtapositional phase. At this location, the algorithm is instructed to conduct local hill-climbing sweeps on as many population members as time permits.

```

Generate Competitive Template
For epoch = 1 to  $j$ 
  For  $n = 1$  to  $k$ 
    Perform Probabilistically Complete Initialization
    Building Block Filtering Phase
    Juxtapositional Phase
  End Loop (for  $n=1$  to  $k$ )
End Loop (for epoch= $1$  to  $j$ )

```

Figure 3.4. Epoch fmGA Algorithm Pseudocode

3.3.4 The Epoch fmGA. Recent work by Knjazew and Goldberg [75] presents a variation to the fmGA which they called an “epoch”. The original fmGA runs through a series of building block sizes for each phase of the fmGA before completing. In their follow-up work, they recommend running through a sequence of building blocks a number of times. The idea is to repeat the original fmGA a number of times. The only modification to the algorithm required is to implement an outer loop around the building block loop (see Figure 3.3.4. Refer to the upcoming chapters to find out the effects of the fmGA with epochs. Other results that combine the epoch concept with population seeding and secondary structure searching can be found in [127, 100].

3.3.5 Searching for Secondary Structures. To improve effectiveness, the algorithm was modified to search for secondary structures. In general, it is not known a priori if whether or not a given secondary structure exists. As a result, the algorithm was modified such that any “known” secondary structure could be incorporated into the algorithm by adding additional angular constraints. This work incorporated only α -helix and beta sheet secondary structure search constraints. A new section of code was added to the algorithm that would essentially take the population set at the end of the Juxtapositional phase and analyze it in an attempt to determine if a certain percentage of those members had α -helix angular values within it. If a certain threshold was met, then a local search was accomplished on that population member with respect to a given secondary structure. Additionally, another search was accomplished if any one of the three ϕ, ψ, ω dihedral angles fell within the constraint, then all the angles would be set to an optimal (mid-center of the range) value and checked to see if the new angles made a difference. Our work [127]

presents additional information with respect to this effort as well as reports some of the results. The next chapter provides further information with respect to the test setup and results. The pseudocode associated with this change is as follows:

The modification of the code, as outlined in the algorithm requires the use of some input parameters (mainly # of Residues, Secondary Structure Percentage). The number of Residues is extremely important to the correct operation of the algorithm as the way the chromosome is constructed to represent the “x” number of dihedral angles for the protein under study and the order in which the dihedral angles appear starting from left to right, are 1st residue (ψ, ϕ, ω), then 2nd, 3rd, etc. The problem with our chromosome definition is that the last residue’s ($\phi, \psi, \text{and } \omega$) dihedral angles are spread out amongst the remaining χ dihedral angles, thus the “Residue -1” restriction.

The Secondary Structure Threshold parameter is required for two reasons. First, so there are as a means to identify the range for each α -helix (ϕ, ψ , and ω) dihedral angles and secondly to determine whether or not a sufficient portion of the population has what appears to be a secondary structure associated with it. As this parameter is increased the search is broadened and the opposite is also true.

3.3.6 Incorporating Ramachandran Constraints into AFIT’s PSP fmGA. Referring back to Section 2.2.9 a considerable amount of work was done by Kaiser and Deerman to incorporate Ramachandran constraint information into AFIT’s GAs. [69, 25] Deerman’s work was most helpful in that it Figure 3.10 merely needed to be copied into the CHARMM energy fitness function that the fmGA uses in place of 3.9. Like Deerman, the code was conditionally incorporated into the algorithm at compile time based on an `#ifdef` construct.

When a chromosome is passed to fitness function *charmm_eval()*, it is first decoded. This decoding process is directly dependent on the encoding of the chromosome in AFIT’s fmGA, as well as other AFIT GA implementations used to optimize the PSP problem. This encoding is implicitly defined in Tables 3.5 and 3.6, and it is different for different molecules.

```

Generate Competitive Template
For epoch = 1 to j
  For n = 1 to k
    Perform Probabilistically Complete Initialization
    Building Block Filtering Phase
    Juxtapositional Phase
    Building Block Analysis
    For p = 1 to pop_size (STEP + 1)
      For q = 1 to chromosome_length (STEP + 10)
        Call Character to Integer Function on (index q + 10 bits)
        Add Results to BBFStorageArray[(q+10)/10]
      End (for q = 1 to chromosome_length)
    End (for p = 1 to pop_size)
    Store Average of BBFStorageArray in AverageBBFStorageArray by
    AverageBBFStorageArray[] := BBFStorageArray[]/(chromosome_length /10)
    ModifiedCompetitiveTemplate = CompetitiveTemplate
    While FoundBetter = 1
      FoundBetter = 0
      For i = 1 to (# of Residues - 1)
        If (AverageBBFStorageArray[i] is within input Threshold Value)
          ModifiedCompetitiveTemplate[i] =  $\alpha$ -helix(i mod 3) dihedral angle
          Evaluate new String
          If ModifiedCompetitiveTemplate  $\neq$  CompetitiveTemplate
            CompetitiveTemplate = ModifiedCompetitiveTemplate
            FoundBetter = 1
          Else
            ModifiedCompetitiveTemplate = CompetitiveTemplate
          End (if ModifiedCompetitiveTemplate ...)
        End (AverageBBFStorageArray[i]...)
      End (i = 1 to (# of Residues - 1))
    End (While FoundBetter)
    FoundBetter = 1
    While FoundBetter = 1
      FoundBetter = 0
      For i = 1 to (# of Residues - 1)
        If (ModifiedCompetitiveTemplate[i] is within input Threshold Value)
          ModifiedCompetitiveTemplate[i] = optimized  $\alpha$ -helix( $\phi, \psi, \omega$ ) dihedral angle
          Evaluate new String
          If ModifiedCompetitiveTemplate  $\neq$  CompetitiveTemplate
            CompetitiveTemplate = ModifiedCompetitiveTemplate
            FoundBetter = 1
          Else
            ModifiedCompetitiveTemplate = CompetitiveTemplate
          End (if ModifiedCompetitiveTemplate ...)
        End (AverageBBFStorageArray[i]...)
      End (i = 1 to (# of Residues - 1))
    End (While FoundBetter)
    FoundBetter = 1
  End Loop (for n=1 to k)
End Loop (for epoch=1 to j)

```

Population Angle Analysis

Identify which Angles met Secondary Structure Constraints
as Defined by Input Parameter

Population Member_i

1	0	1	1	0	1	1	0	1	1	1	1	1	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



-57.21	-47.39	179.39	-51.39	89.39	-53.39
--------	--------	--------	-------	--------	-------	--------

Φ	Ψ	ω	-	-	-
--------	--------	----------	-------	---	---	---

Repeated for n-1 Residues (Met 4, Poly 13)

23	13	22	-	-	-
----	----	----	-------	---	---	---

Angular Analysis of All Population Members After Juxtaposition Phase
Number Represents Total # of Angles that Fell within Φ , Ψ , ω Constraints

Figure 3.6. Population Angle Analysis

Population Angle Analysis

1st Attempt at Locating a Secondary Structure

Current Competitive Template (CT) Dihedral Angular Values

-45.33	-42.39	78.39	-67.88	90.88	-57.21
--------	--------	-------	-------	--------	-------	--------

If Φ dihedral falls within input parameter constraint, change corresponding CT angle to optimum Φ angle, evaluate, if better keep, else change back
Do previous step with Ψ and ω dihedral angle for n-1 residues

23	13	22	-	-	-
----	----	----	-------	---	---	---

Y	N	Y	-	-	-
---	---	---	-------	---	---	---

-57.21	-42.39	180.00	-67.88	90.88	-57.21
--------	--------	--------	-------	--------	-------	--------

New Competitive Template Angles

Figure 3.7. Population Angle Analysis - 1st Search

Table 3.5. [Met]-Enkephalin Encoding Scheme

Amino Acid	Tyr	Tyr	Tyr	Gly ₁	Gly ₁	Gly ₁	Gly ₂	Gly ₂	Gly ₂	Phe	Phe	Phe
Dihedral Angle	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω
Starting Bit in the Chromosome	0	10	20	30	40	50	60	70	80	90	100	110
Amino Acid	Met	Tyr	Tyr	Phe	Phe	Met	Met	Met	Met	Tyr	Met	Met
Dihedral Angle	Φ	χ_1	χ_2	χ_1	χ_2	χ_1	χ_2	χ_3	Ψ	χ_3	χ_4	ω
Starting Bit in the Chromosome	120	130	140	150	160	170	180	190	200	210	220	230

Population Angle Analysis

2nd Attempt at Locating a Secondary Structure

Current Competitive Template (CT) Dihedral Angular Values

-57.21	-42.39	180.00	-67.88	90.88	-57.21
--------	--------	--------	-------	--------	-------	--------

If a percentage of Φ , Ψ and ω dihedral within a single residue falls within input parameter constraint, change the other one or two angles in the CT
 Φ , Ψ or ω angle(s), evaluate, if better keep, else change back

Do previous step with remaining n-1 residues

-57.21	-47.33	180.00	-67.88	90.88	-57.21
--------	--------	--------	-------	--------	-------	--------

New Competitive Template Angles

Figure 3.8. Population Angle Analysis - 2nd Search

Table 3.6. *Polyalanine*₁₄ Encoding Scheme

Amino Acid	Ala ₁	Ala ₁	Ala ₁	Ala ₂	Ala ₂	Ala ₂	Ala ₃	Ala ₃	Ala ₃	Ala ₄	Ala ₄	Ala ₄
Dihedral Angle	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω
Starting Bit in the Chromosome	0	10	20	30	40	50	60	70	80	90	100	110
Amino Acid	Ala ₅	Ala ₅	Ala ₅	Ala ₆	Ala ₆	Ala ₆	Ala ₇	Ala ₇	Ala ₇	Ala ₈	Ala ₈	Ala ₈
Dihedral Angle	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω
Starting Bit in the Chromosome	120	130	140	150	160	170	180	190	200	210	220	230
Amino Acid	Ala ₉	Ala ₉	Ala ₉	Ala ₁₀	Ala ₁₀	Ala ₁₀	Ala ₁₁	Ala ₁₁	Ala ₁₁	Ala ₁₂	Ala ₁₂	Ala ₁₂
Dihedral Angle	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω	Φ	Ψ	ω
Starting Bit in the Chromosome	240	250	260	270	280	290	300	310	320	330	340	350
Amino Acid	Ala ₁₃	Ala ₁₃	Ala ₁₃	Ala ₁₄	Ala ₁	Ala ₂	Ala ₃	Ala ₄	Ala ₅	Ala ₆	Ala ₇	Ala ₈
Dihedral Angle	Φ	Ψ	ω	Φ	χ_1	χ_2	χ_3	χ_4	χ_5	χ_6	χ_7	χ_8
Starting Bit in the Chromosome	360	370	380	390	400	410	420	430	440	450	460	470
Amino Acid	Ala ₉	Ala ₁₀	Ala ₁₁	Ala ₁₂	Ala ₁₃	Ala ₁₄	Ala ₁₄					
Dihedral Angle	χ_9	χ_{10}	χ_{11}	χ_{12}	χ_{13}	Ψ	χ_{14}	ω				
Starting Bit in the Chromosome	480	490	500	510	520	530	540	550				


```

indexPtr = Indep_dihedral;
while (indexPtr != NULL)
{
    temp = (double) Ctoi (&buff[start],
slice);
    P[n] = ((temp / max_range) * twoPI) -
PI;
    n++;
    start = start + slice;
    indexPtr = indexPtr->next;
}

```

Figure 3.9. Original Ramachandran Decoding

In the previous implementation, each dihedral angle is translated from its 10-bit binary encoding to a radian value (a C double) between 0 and 2π . In the new constrained CHARMM decoding, the dihedral is decoded and then mapped to the appropriate constrained subrange of possible values depending on which dihedral it represents.

3.3.7 Short-circuiting the Energy Fitness Function. Overall the computational cost associated with the fitness function is a major factor during runtime of the fmGA. Deerman did show that the various subfunctions of AFIT's CHARMM implementation had a cost difference as high as 10^4 in magnitude ([25] Figure 43) and this research effort confirmed his results.

The idea of attempting to short-circuit the energy fitness function was formulated during a review of a paper about long-distance interactions between atoms. [85] Figure 3.11 previously shown in Chapter 2 under the discussion of energy functions shows that AFIT's current fitness function is really a summation of a number of subfunctions, each of them a summation of a unique physical property associated with the nuclear forces found at the atomic level. Since the derived fitness value is associated with a particular solution string and subsequently decisions are made based upon those fitness values, so long as a comparison between the same contributing factors of the fitness value exist, it was thought that comparison between two solutions would still be relevant. The first term was used as the location to cutoff the fitness function. This was chosen as it was

```

indexPtr = Indep_dihedral;
while (indexPtr != NULL) {
    temp = (double) Ctoi (&buff[start], slice);
    temp = ((temp / max_range) * twoPI) - PI;
    #ifdef Met-Enkaphalin
    switch (n)
    /* non-glycine phi */
    case 1: case 10: case 13:
        temp = temp*((-210 - -30)/twoPI) + -210;
        break;
    /* glycine phi */
    case 4: case 7:
        temp = temp*((-315 - -45)/twoPI) + -315;
        break;
    /* psi */
    case 2: case 5: case 8: case 11: case 21:
        temp = temp*((210 - -90)/twoPI) + 210;
        break;
    /* omega */
    case 3: case 6: case 9: case 12: case 24:
        temp = temp*((-160 - -200)/twoPI) + -160;
        break;
    /* chi1&3 tyrosine, chi2&3 methionine */
    case 14: case 22: case 19: case 20:
        temp = temp*((-160 - -200)/twoPI) + -160;
        break;
    /* chi2 tyrosine, chi1 pheny, chi1 methionine */
    case 15: case 16: case 18:
        temp = temp*((75 - 45)/twoPI) + 75;
        break;
    /* chi2 pheny, chi4 methionine */
    case 17: case 23:
        temp = temp*((-75 - -45)/twoPI) + -75;
        break;
    default: printf("error in n = %i",n); exit(1);
    #endif
    P[n] = temp;
    n++;
    start = start + slice;
    indexPtr = indexPtr->next; }

```

Figure 3.10. Deerman's Modified Ramachandran Decoding [25]

$$E_{total} = E_b + E_{\Theta} + E_{\Phi} + E_w + E_{vdW} + E_{el} + E_{hb} + E_{cr} + E_{c\Phi}$$

Figure 3.11. CHARMM Energy Model

```

func(F)
{
    energy = fixed_energy; /* start with the fixed amount of energy */
                          /* then add the energy due to dependent */
                          /* and independent bonds, angles, and */
                          /* dihedrals, as well as non-bonded and */
                          /* improper dihedral energy */
    energy += non_bonded_energy(indexPtr, 1.0);
    energy += non_bonded_energy(indexPtr, 0.5);
    energy += bond_energy(indexPtr->i, indexPtr->j, sqrt(bond_length),
                          indexPtr->k_parm, indexPtr->bodanglephi);
    energy += bond_energy(indexPtr->i, indexPtr->j,
                          atom[indexPtr->i].bond_length,
                          indexPtr->k_parm, indexPtr->bodanglephi);
    energy += angle_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                          bond_angle, indexPtr->k_parm, indexPtr->bodanglephi);
    energy += angle_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                          atom[indexPtr->i].bond_angle, indexPtr->k_parm,
                          indexPtr->bodanglephi);
    energy += dihedral_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                              indexPtr->l, dihedral_angle, indexPtr->k_parm,
                              indexPtr->bodanglephi, indexPtr->n_or_phase);
    energy += dihedral_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                              indexPtr->l, atom[indexPtr->i].dihedral_angle,
                              indexPtr->k_parm, indexPtr->bodanglephi,
                              indexPtr->n_or_phase);
    energy += improper_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                              indexPtr->l, dihedral_angle, indexPtr->k_parm,
                              indexPtr->bodanglephi);
    return (energy);
}

```

Figure 3.12. Sub-component Summation Code of AFIT's CHARMM Energy Fitness Function

demonstrated through empirical testing that the first term tended to add an inordinate amount of energy (as high as 10^9 kcal) to the overall fitness value if the solution string was not good. It was assumed that the target protein's energy should be at least less than 1000kcal (for the [Met]-Enkephalin, the accepted energy fitness value is approximately -30kcal) the algorithm was modified such that if the first subcomponent of the CHARMM energy fitness function resulted in a total energy of more than 1000kcal, then the algorithm would immediately terminate and return that value to the calling procedure.

Figure 3.12 is an extraction of the actual C code summing up the individual subcomponents that comprise the energy fitness function; however, most of the code was removed with the exception of the various components of the energy function and their ordering of their contribution to the overall energy fitness value. In the original code, once all the subfunction contributions have been added to the variable *energy* then *energy* is returned to the calling procedure. The code was modified to reflect the changes this short-circuiting idea required and can be seen in Figure 3.13.

```

func(F)
{
    energy = fixed_energy; /* start with the fixed amount of energy */
                          /* then add the energy due to dependent */
                          /* and independent bonds, angles, and */
                          /* dihedrals, as well as non-bonded and */
                          /* improper dihedral energy */
    energy += non_bonded_energy(indexPtr, 1.0);
    {
        if (energy > 1000)
            return (energy)
        else
        {
            energy += non_bonded_energy(indexPtr, 0.5);
            energy += bond_energy(indexPtr->i, indexPtr->j, sqrt(bond_length),
                                indexPtr->k_parm, indexPtr->bodanglephi);
            energy += bond_energy(indexPtr->i, indexPtr->j,
                                atom[indexPtr->i].bond_length,
                                indexPtr->k_parm, indexPtr->bodanglephi);
            energy += angle_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                                bond_angle, indexPtr->k_parm, indexPtr->bodanglephi);
            energy += angle_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                                atom[indexPtr->i].bond_angle, indexPtr->k_parm,
                                indexPtr->bodanglephi);
            energy += dihedral_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                                indexPtr->l, dihedral_angle, indexPtr->k_parm,
                                indexPtr->bodanglephi, indexPtr->n_or_phase);
            energy += dihedral_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                                indexPtr->l, atom[indexPtr->i].dihedral_angle,
                                indexPtr->k_parm, indexPtr->bodanglephi,
                                indexPtr->n_or_phase);
            energy += improper_energy(indexPtr->i, indexPtr->j, indexPtr->k,
                                indexPtr->l, dihedral_angle, indexPtr->k_parm,

```

Figure 3.13. Modified Sub-component Summation Code of AFIT's CHARMM Energy Fitness Function

Although the changes incorporated into the algorithm were based entirely on empirical results for the Met-Enkephalin protein, the concept does hold merit in that with further study and advice from the experts, the subfunctions that comprise the CHARMM energy fitness function could be rearranged such that the most contributing portions could be placed ahead of all the other lesser contributing factors.

3.3.8 Decimal Representation of Dihedral Angles as Output. This change in code is merely a cosmetic addition in that it provides the user a more direct method in which to associate the 240 bit stream of 1's and 0's (binary representation) of all dihedral angles with a decimal representation. The output is modified in the show_string function and utilizes the existing ctoi (Character to Integer) function already incorporated in the code. Once each set of 10 bits are converted to an integer, its results are multiplied by (1024/360) and that result is subtracted from 360 yielding the actual dihedral angle in decimal form (refer to Equation 3.1).

$$D_d = 360 - \left\lceil \frac{1024}{360} * ctio(D_b) \right\rceil, \quad (3.1)$$

where

D_d = Dihedral Angle (decimal representation)

D_b = Dihedral Angle (binary representation)

ctio() = Character-to-Integer Function

3.3.9 Scaling the fmGA to larger Proteins. The scaling of the algorithm to a larger protein is inherently accomplished by the incorporation of the aforementioned code changes. It is hoped that these enhancements improve the effectiveness and efficiency with respect to the Met-Enkephalin and provide favorable results for a protein that is 2.5 times larger in size. The major changes in code, those dealing specifically with the incorporation of additional secondary structure, play a major role in aiding the fmGA in finding a “semi” optimal solution to the PSP problem with respect to larger proteins. As indicated previously the solution space increases tremendously with only a small increase in the number of dihedral angles defined in the chromosome string and without these changes little hope of scaling the fmGA to even larger-sized proteins is warranted.

3.4 Summary

Software reuse is a major design decision for genetic algorithm implementations at AFIT. The designs presented in this chapter were implemented by Gates [41] and his work with the fmGA is the main contributor to this research. Both Merkle and Gates led the charge in parallelizing the fmGA using both NX and MPI communication calls. [93] Credit should be given to both and not attributed to this work. The only design changes that were made to the code were in the areas mentioned in Section 3.2.3. The changes do not change the work of both Gates or Merkle, but they do improve upon them. The pseudocode representation was used as the changes do not change the operation of the algorithm, with respect to space/time operation of the algorithm. The next chapter describes the test

setup to measure performance characteristics of these changes with respect to the original fmGA and its parallelized version that Gates and Merkle developed.

IV. Experimental Design

4.1 Introduction

Before the advent of the “Scientific Method”, many would-be researchers merely used the trial and error method in an attempt to gain insight into a particular problem. The scientific method is the process by which scientists, collectively and over time, endeavor to construct an accurate (that is, reliable, consistent and non-arbitrary) representation of the world or the problem which they study. Recognizing that personal and cultural beliefs influence both our perception and our interpretation of natural phenomena, we aim through the use of standard procedures and criteria to minimize those influences when developing a theory. In summary, the scientific method attempts to minimize the influence of bias or prejudice in the experimenter when testing an hypothesis or a theory. For an in depth look at developing scientific experiments the reader is referred to [2]. For the purpose of this research, the observed phenomenon is the protein structure prediction problem, and the hypothesis is that the modifications made to the fmGA, as developed by Gates and Merkle, results in an improvement in overall effectiveness and efficiency of that algorithm. This chapter discusses how to test various modifications to the fmGA described and developed in the previous chapter.

Many papers appearing in journals reporting computational experiments use computer generated evidence to compare or rank competing mathematical software techniques. Unfortunately, very little effort in establishing general guidelines indicating how the experiments should be conducted or how the results should be presented exist. We have provided four references which we use to direct our experimentation [60, 123, 143, 126]. According to Jackson, et.al., the researcher should always keep in mind the items identified in Table 4.1.

Professor Tufte, a Yale professor and the award-winning author of three books on information design, provides a one-day course titled “Presenting Data and Information” [143]. The attendees are inundated with several information designing topics listed in Table 4.1. His attempt at covering a broad range of topics in this area is quite successful and

Table 4.1. What to Keep In Mind When Testing?

1. Are the results presented sufficient to justify the claims made?
2. Is there sufficient detail to reproduce the results?
3. When should a statistically-based experiment be done—usually when a claim such as “this method is better (i.e. faster, more accurate, more efficient, easier-to-use, etc.)”?
4. Are the proper test problems being used?
5. Are all possible performance measures (efficiency, robustness, reliability, ease-of-use) addressed?
6. Is enough information provided with respect to the architecture of the hardware being used?

Table 4.2. Professor Tufte’s One-day Course on Presenting Data and Information

1. Fundamental Strategies of Information Design
2. Color and Information
3. Statistical Data: Tables, Graphics, and Semi-Graphics
4. Business, Scientific, research and financial presentations
5. Complexity and Clarity
6. Effective Presentations: on paper or in person
7. Use of Video, Overheads, Computers, and Handouts
8. Courtroom Exhibits
9. Design of Information Displays in Public Schools
10. Animation and Scientific Visualizations
11. Design of Computer Interfaces and Manuals

although the cited reference is not very detailed, it did provide general direction for this document and how to properly present information.

Finally, this chapter is organized such that the first section discusses input and output data, the identification of all parameters available during testing (for all tests they are the same unless otherwise indicated), a discussion of the random number seed and other topics that are pertinent to the set of experiments. Following the general information, each individual experiment is presented with the objective and methodology of the experiment identified. For each experiment any parameter settings or environmental settings that differ from the generalized discussion are duly noted.

4.2 General

The following subsections provide details about various criteria that are common to the subsequent set of experiments discussed in this chapter. We review the proteins, the input/output data, random number seeds, root mean square deviation, test platforms, and experiment specifics (parameter values) for the set of experiments discussed after this section.

4.2.1 Test Proteins. The proteins used for this research are the Met-[Enkephalin] and the model Polyalanine₁₄. Alternative molecules have been considered, (e.g., Crambin [132], P27-4, P27-6, and P27-7 [118], and others mentioned in Deerman's thesis [25]); however they were not used for three reasons; first, they are considerably larger (thousands of atoms as opposed to 75 and 143 for the [Met]-Enkephalin and Polyalanine₁₄, respectively) than the [Met]-Enkephalin and Polyalanine₁₄ proteins (which would have required an exponential increase in computational power), they have no accepted minimum conformation at this time, and additional steps to convert the source files to a standard that can be used by AFIT's CHARMM implementation is under review and was not readily available.

4.2.1.1 Met-Enkephalin and Polyalanine. In previous PSP research at AFIT the protein most commonly used is the pentapeptide [Met]-Enkephalin (see Figure 2.2). It is a relatively small and simple protein defined by the five-amino-acid sequence Tyr-Gly-Gly-Phe-Met using neutral NH₂ and -COOH as terminators at the α -amino and α -carboxyl ends, respectively. It is principally composed of carbon (C), oxygen (O), and nitrogen (N) atoms. The two principal factors influencing the selection of this particular protein for study are; first, its native conformation is known, and secondly, other researchers have used energy minimization to predict its tertiary structure. [43, 88, 25, 28, 69, 41]

The second molecule, Polyalanine₁₄, was chosen because of its affinity to a nicely folded α -helix structure. Polyalanine₁₄, a larger polypeptide than [Met]-Enkephalin, is defined by 14 amino-acid groups: *Ala*₁-*Ala*₂-*Ala*₃-...-*Ala*₁₄. We have chosen to use the same end groups as the [Met]-Enkephalin molecule. Figure 2.2 and Figure 2.3 are representations

of [Met]-Enkephalin and Polyalanine₁₄, respectively. The figures are labeled to distinguish the dihedral angles along their molecular backbone.

Tables 2.1 and 2.2 indicate the “correct” dihedral angles values for the “accepted ” energy minimum defined by QUANTATM. The conformation energy for [Met]-Enkephalin is -29.225.

4.2.2 General Data Requirements.

4.2.2.1 Input Data. The only input data required for the modified fmGA to work properly is the standard “Z-matrix” and “RTF” files [4] (associated with the Met-Enkephalin and Polyalanine proteins) that were used in the original version of the fmGA as well as other genetic algorithms implemented at AFIT. A discussion of these files can be found in prior AFIT researcher’s efforts [41, 4, 25, 69, 43] in their respective appendices.

4.2.2.2 Output Data. Table 4.3 provides a list of the various types of data that were obtained through the entire run of the program and captured to a local file. The different types of data were important for the various reasons indicated. It is important to note that not all data was used in every experiment and only those data that were pertinent to a given experiment’s intent were used.

4.2.3 Random Number Seed. In Deerman’s work [25], a lengthy discussion of his efforts to capture the effects of choosing a random number seed is discussed. In Chapter 2 and Appendix I of this document, an additional discussion of Random Number Generators can be found. For all experiments, the fmGA algorithm has an initial starting seed of 123456789 and every subsequent execution of the algorithm (depending on the number of experiments to be conducted) takes the current seed and alters it by Equation 4.1.

$$Seed = ((Seed * ((Experiment + 1) * (NodeId + 1)) * PRIME) \& MASK) \quad (4.1)$$

Table 4.3. fmGA Output Data

Data Category	Description	Purpose	Experimental Purpose
Best Individual's Fitness	Fitness of the best individual from each experiment.	This is one indication of the performance of a GA.	Effectiveness
Best Individual's Chromosome	The chromosome of the best individual from each experiment.	The chromosome indicates the individual's location within the search area.	Visualization
Best Individual's Coordinates	The coordinates of the best individual from each experiment.	The coordinates of each of the best individuals will be analyzed using RMS to determine the accepted energy minimum configuration.	Effectiveness
Worst Individual's Fitness	Fitness of the worst individual from each experiment.	This is one indication of the performance of a GA.	Effectiveness
Average Fitness	The average fitness of the population from each experiment.	An indication of the overall performance of the GA.	Effectiveness
Parameters	All parameters manipulated within the GA.	Supplies all the information required to regenerate each experiment.	Historical Information
Execution Time	The total time for each experiment.	Used to indicate scalability and efficiency of the approaches.	Efficiency
The number of fitness evaluations.	The number of times the CHARMm is called.	A commonly used indication of efficiency.	Efficiency
Best Fitness per Block Size	Best Fitness after each successive Building Block Iteration	Used to examine searching capability of fmGA	Effectiveness

where Seed is the Seed to start the Random Number Generator

Experiment is the current iteration of experiments

NodeId is the node the algorithm is running on (used in parallel implementations)

PRIMER is equal to 65539

MASK is equal to equation 4.2

$$(0 << (INTSIZE - 1)) \quad (4.2)$$

where INTSIZE is 32

The equation is such that it assigns different seeds to different nodes for each successive experiment. While this portion of the code wasn't modified, there may be an issue associated with the fact that there is a connection between successive seeds being gener-

ated on a particular node (refer to the section on Random Number Generators in Chapter II and Appendix I). Further investigation is required.

4.2.4 Root Mean Squared Deviation (RMSD). Deerman, Gates, and Kaiser talks about using a RMSD method to determine the effectiveness of the results obtained [25, 41, 69]. Root Mean Squared Deviation (RMSD; $p=2$ norm) is one way of comparing a calculated molecule's conformation to the naturally occurring conformation independent of the calculated fitness. Other norms include p -norm, maximum distance between points, maximum difference in probability, absolute difference etc.; however, he chose to use this particular method to determine the difference between the known and calculated geometrical configuration. For instance, a GA may produce a molecule conformation (X), which approximates the accepted energy level of the best known conformation (Y), but internal coordinates for each atom within X may not closely correspond to the positions within Y. Thus, this analysis compares a GA's best produced molecule to the naturally occurring molecule. RMSD is used in particular because the PSP community commonly references RMSD calculations. Therefore, we can compare our results to other research efforts, although, it must be stressed that different energy models may provide different energy values for the same molecule even when the internal geometry is the same. The general equation for this calculation is:

$$RMSD = \sqrt{\sum_{i=1}^{number\ of\ dihedral\ angles} (dihedral_{known} - dihedral_{calculated})^2} \quad (4.3)$$

In the final section of the next chapter, we look at alternative ways to evaluate given proteins with respect to three different RMSD methodologies and select one to compare our results with previous work and that of the accepted solution.

4.2.5 Test Platforms. Experiments are executed on the Aeronautical Systems Center (ASC) Major Shared Resource Center (MSRC) IBM SP2 & SP3; the Air Force Institute of Technology (AFIT) cluster of workstations (COW); and on the AFIT ABC Beowulf. After initial testing, it was decided not to use the COW due to its lack of computational power as the current configuration contains mostly processors with speeds

Table 4.4. Parametric Settings for Most Experiments

Parameter	Description	Impact Upon GA
Seed	987654321	YES
Building Block (min-max)	6-10 Met-Enkephalin 16-20 Polyalanine	YES
Experiments	10	YES
String Length	560 Polyalanine 240 [Met]-Enkephalin	YES
Genic Alphabet	01	YES
Shuffle Number (>1)	2	YES
Cut Probability	0.02	YES
Splice Probability	1.0	YES
Overflow (>1.0)	1.6	YES
Sweeps	Most cases 3; however it depends on the test	YES
Competitive Template Guesses	Most cases 5; however it depends on the test	YES
Number of Residues	5 [Met]-Enkephalin 14 Polyalanine	YES
Protein Structure Used	Met Poly	YES
Heterogeneous	No; however it depends on the test	YES
Initial Energy Cutoff Value	500 [Met]-Enkephalin 800 Polyalanine	YES
Secondary Structure Percentage	Most cases 0; however it depends on the test	YES
Good Population Percentage	Most cases 0; however it depends on the test	
Population Recombination	I	YES
Initialization Flag	C	YES
Distinct Competitive Templates	Most Cases No; however it depends on the test	YES
Global Selection	N	YES
Migration	N	YES
Minimization Probability	0.02	YES
Replacement Probability	0.01	YES
Primordial Generations	200	YES
Total Generations	400	YES
n_a	Most cases 50; however it depends on the test	YES
BBF Schedule	See other tables in this section	YES

of less than 300MHz. Hardware limitations dictate the current upper limit for the AFIT Beowulf system. For a complete description of each system, see Appendix B.

4.2.6 Experiment Specifics. Unless otherwise indicated, each experiment used the parametric values found in Table 4.4. It should be noted that the Input Schedule for the [Met]-Enkephalin (Table 4.5) and Polyalanine₁₄ (Table 4.6) are distinct from each other and while the [Met]-Enkephalin Input Schedule was taken directly from Gates' and Merkle's work [41, 93], the Polyalanine₁₄ Input Schedule was created to reflect that of the [Met]-Enkephalin. Additionally, both theoretical and empirical work in this area is lacking which means that it very likely that the input schedules used are not the best for this particular problem. In fact, it is highly probable that the input schedule must be changed to reflect the problem that is being addressed.

Table 4.5. Input Schedule for the Polyalanine Protein

Cut_gen	Str_len	Threshold
0	540	500
7	480	460
11	440	420
15	400	380
19	380	360
23	360	360
29	340	320
35	300	280
41	260	240
47	220	200
53	180	140
59	148	128
65	125	105
71	109	99
77	94	83
82	81	67
87	68	56
92	55	45
97	43	34
102	30	24
107	20	14
112	19	13
117	18	12
122	17	11

Table 4.6. Input Schedule for the [Met]-Enkephalin Protein

Cut_gen	Str_len	Threshold
0	216	194
7	185	143
11	157	107
15	135	84
19	115	64
23	98	47
29	84	38
35	72	31
41	61	25
47	53	21
53	45	17
59	39	15
65	33	12
71	29	10
77	24	8
82	21	7
87	18	6
92	15	5
97	13	4
102	11	4
107	9	3
112	8	3
117	7	2
122	6	2
127	5	3

Table 4.7. List of Experiments

1. Parallel vs Serial fmGA (minor)
2. Generating Competitive Templates with the Hill-Climbing Sweep Function (minor)
3. Random Search with and without the Hill-Climbing Sweep Function (minor)
4. Improving Performance in a Heterogeneous Environment (MAJOR)
5. Searching for a Secondary Structure (MAJOR)
6. Effects of Seeding the Population (MAJOR)
7. Incorporating the Epoch Idea (minor)
8. Short-circuiting the Energy Fitness Function (minor)
9. Effects of Ramachandran Constraints (minor)

Table 4.8. Possible Experiments Not Conducted During this Research

1. Conduct Experiments with other Proteins (Crambin, larger Polyalanine proteins, etc.)
2. Incorporate All Code Modifications and Experiment
3. Exhaustive Testing to Determine the Best Possible Parametric Values
4. Conduct Production Test on 128 nodes with Large Population Size To Obtain Best Answer

4.3 Statistical Testing and Techniques

As indicated previously the RMSD test is used to determine our best results from that of the optimal and previous results obtain by Gates [41]. Additionally, the Kruskal Wallis test is used in some of the experiments and is clearly marked where used. Details about the RMSD and Kruskal Wallis are provided in section 5.11 of the following chapter.

4.4 Experiments

The set of experiments is separated into two categories—major and minor. The two categories reflect the overall importance they have to this research. Major experiments focus on the various efforts to improve the effectiveness of the fmGA with domain specific information or radical changes to the fmGA to improve overall performance. Minor experiments are all other experiments. Table 4.7 identifies all the experiments and the category to which they belong.

Table 4.8 provides a simple list of other experiments that were not accomplished due to lack of time or other reasons.

4.4.1 Experiment 1: Parallel vs Serial fmGA. Objective: The purpose of this experiment is twofold. The first objective is to determine the effectiveness of the parallel version of the fmGA (pfmGA) with respect to the population size and number of nodes in the parallel run. The other objective is to characterize efficiency of the pfmGA in terms of overhead, speed-up, and scalability.

Methodology: Goldberg's serial implementation [46], which Gates blended with AFIT's CHARMM energy model is used for the pfmGA [41]. A data parallelized implementation based upon the sequential fmGA is employed for the multi-node experiments. The previous chapter provided low-level design and implementation details. Parallel executions using 1, 2, 4, 8, 16, 32, 64, and 96 processors are used to evaluate scalability and speed-up of this algorithm. Parameters for each test case are supplied at the beginning of this chapter. Results and analysis are presented in Section 5.2.

4.4.2 Experiment 2: Generating Competitive Templates with the Hill-Climbing Sweep Function. Objective: The purpose of this experiment is again twofold; first, to determine the effect of the hill-climbing sweep method that is used in the fmGA to find a *good* initial competitive template. Since the number of sweeps is an input parameter and no prior research has been done to determine what value should be assigned, the intent of this experiment was to determine if the sweeping mechanism was doing extra work for little or nothing in improved performance (depending on the value of the input parameter). The other objective is probably more important and that is to determine through empirical results the efficiency vs effectiveness of the sweeping mechanism in deriving a good initial competitive template.

Methodology: The hill-climbing sweeping mechanism essentially is a sweep through the chromosome switching a bit and evaluating the new chromosome. If the result of the change is better then it is kept, else, it is changed back to what it was before and the algorithm proceeds on the sweep stops when no better solution is found within one iteration

of the sweep function [71]. The original code would allow the algorithm to continue one until the input parameter (Sweeps) had been met. The series of tests are accomplished by changing the input parameters Sweeps and Number of Competitive Templates to 1, 2, 3, 5, 10, and 15, respectively. The tests were run on a single 600MHz system and only against the Polyalanine protein. The only output data collected are the resulting fitness values and the time it took to complete as the intent of this experiment is ascertain the effects of the code changes with respect to fitness vs time. Results and analysis are presented in (Section 5.3).

4.4.3 Experiment 3: Random Search with and without the Hill-Climbing Sweep Function. Objective: The purpose of this experiment is to determine if a random search (either with or without a hill-climbing optimization technique) can out perform a fmGA.

Methodology: To effect the random search with the hill-climbing sweep function we have the fmGA algorithm generate 250 competitive templates and sweep (conduct a localized search) each competitive template until no better solution is found. The value of 250 is used as it has a near equivocal time associated with it and some other serial runs of the fmGA. In order to generate 100,000 random points for the second set of tests, we generate 100,000 competitive templates without any sweeps (essentially a random population member). Both methods are allowed to run for an approximately the same time it takes a serial run of the fmGA on the Met-Enkephalin. In this experiment, the Sweep input variable is set to max to allow the sweeping function to continue until no better solution is found or zero to generate a completely random solution. For both tests, all generational values are set to 0 and other parameters are set to essentially turn off their functionality during program execution. Tests are done in serial mode only with the results and analysis presented in Section 5.4.

4.4.4 Experiment 4: Improving Performance in a Heterogeneous Environment. Objective: The purpose of this experiment is to analyze the efficiency and effectiveness of the pfmGA in a heterogeneous parallel platform. A secondary objective of this test is to determine the best memetic approach or local search method to use with this realized

computational power. There were two local search methods that were looked at (Baldwinian and Lamarkian) searches. Lamarkian is simply conducting a localized search and replacing the original chromosome (solution string) and fitness value with the final solution string and its associated fitness value. Baldwinian method is where a localized search is done but instead of replacing the chromosome, only the final fitness value is kept. This method retains the original chromosome string but it has a fitness value associated with it that indicates its potential. The tests were conducted on 12 machines with a range of processing speed from 1GHz to 350MHz. The diversity, with respect to processor speed is used to emphasize the differences between them which in turn demonstrates the amount of computational power being wasted with the original pfmGA algorithm.

Methodology: The algorithm is modified such that those machines which are faster than the slowest machine in a set of heterogeneous machines (with respect to raw processing power), are allowed to do extra work. The amount of time available to do extra work is calculated by Equation 4.4. The extra work to be done is located in two different portions of the pfmGA code (see Figure 3.3.3). The first location is after the algorithm generates a competitive template. The faster machines are allowed to generate a number of extra competitive templates based on the amount of extra time the specific processor has available. The second location is at the end of a building block iteration. The faster machines are allowed to conduct local searches on the population members in the extra time they have before the slowest machine reaches that point.

$$T_{AV(j)} = T_{[\min_{(i=0..n)}(node_i)]} - T_{node_j} \quad (4.4)$$

where $T_{AV(j)}$ = Time Available on $node_j$

$T_{[\min_{(i=0..n)}(node_i)]}$ = Time of slowest machine

T_{node_j} = Time of this machine

n = Number of Nodes being used minus 1

In general, most heterogeneous computing systems are made up of individual computers that are tied together by some local area network. More often than not the collection of systems are dependent upon the budget of the department setting up and using the

parallel system. It is often the case that the creation of the parallel system is done in incremental steps with the idea of defraying the overall cost over a number of fiscal years. As a result we often see parallel systems with a wide range of individual systems. This common situation is what drove the development of AFIT's Pile of PCs to its current processor configuration today. See Table B.3 for a complete list of the processors included in AFIT's Pile of PCs. With this in mind and because all processors were not available during testing, all tests were conducted on 12 Intel Pentium II and III machines which consisted of five 1GHz MHz machines, six 600 MHz machines, and one 350 MHz machine. This wide range of machines was utilized to illustrate the sheer number of cycles that could be wasted in a heterogeneous environment where the differences in machine speeds is quite significant.

Note: While it would have been nice to use all the available machines, it is not necessary to determine the effects of the code modifications. What is required is a significant difference in the processing capability of two or more machines. The algorithm determines what "significant difference" is required in order for the observer to observe the difference. In other words, if the processor speeds are very similar, then very little benefit or utilization of wasted speed will be noticed. Thus, it is the difference in processor speed that will help the observer recognize the effects of the algorithm.

Initially the intent was to run the modified algorithm three separate times, once for the base case, once for the Lamarkian case and finally once for the Baldwinian case; however, after the initial run with the Lamarkian it was noticed immediately that the results obtained did not contain good fitness values. As a result, only two of the three cases were actually run. Further explanation as to why the results were not good for the Lamarkian case are provided in the following chapter's, Section 5.5. The results are compared with those of a fmGA starting with the same seed to initialize the random number generator. Results and analysis are presented in Section 5.5.

4.4.5 Experiment 5: Searching for a Secondary Structure. Objective: The purpose of this experiment is to determine if searching for a secondary structure can provide better results (improve effectiveness). The test was conducted on both proteins because the

[Met]-Enkephalin doesn't contain any known secondary structure and the Polyalanine₁₄ is known for its right-handed α -helix secondary structure. The test is run in parallel on seven machines (5 1GHz machines and 2 933MHz machines) as they were the fastest machines available and the slower set of machines would have increased the runtime by a factor of 2 or more. The tests are run through a series of different parametric values.

Methodology: A new input parameter is used to set the constraints around the *optimal* ϕ, ψ, ω dihedral angle values for the α -helix secondary structure (see Figure 3.3.5). While there were a large number of ways to test, we decided to test the code modifications in the following manner. For the first set of the experiments the range of that parameter was from 0% to 100% (in increments of 5% until 60% then in increments of 10%). The second set of experiments were accomplished by changing the *n.a* parameter which directly affects the population size to (100, 200, 400, and 800) and holding the Secondary Structure parameter constant to 100%. These particular test method were used as it allowed us to cover the full range of the input parameter and changing the population size allowed us to ascertain the effect on timing as well as effectiveness for an exponentially increasing workload size. Results and analysis are presented in Section 5.6.

4.4.6 Experiment 6: Effects of Seeding the Population. Objective: The purpose of this experiment is to determine the effects of seeding the initial population with a percentage (new input parameter) of members that have secondary structure (α -helix and β -sheet) related information, with locally optimized population members, or a combination of them. Both the effectiveness and efficiency of this seeding is analyzed. The experiment was tested on a single machine only due to the extensive set of tests and the length that each took (time-wise this series of tests is the longest run as the above two sets of tests are run against both proteins).

Methodology: The algorithm is modified such that it creates a given percentage of the population according to either the Alpha, Beta, Swept, or Combo setting of the input parameter. ψ, ϕ , and ω dihedral angular values for α -helix and β -sheet, used to determine the center of the constraint, are taken from Table D.3 and are randomly selected. There are four sets of tests for each of four seeding methods (Alpha, Beta, Swept, and Combo) each

initializing the population with either (0, 5, 10, 20, 30, 40, 50, or 100%) of the population initialized according to the method under testing. Results and analysis are presented in Section 5.7.

4.4.7 Experiment 7: Incorporating the Epoch Idea. Objective: The purpose of this experiment is to determine the effects of incorporating the idea of wrapping the fmGA block-by-block iteration with an outside loop known as an epoch. Both the effectiveness and efficiency aspects are evaluated. The experiment is only run on a single processor as additional processors could potentially mask the epoch results.

Methodology: The algorithm is modified such that it contains an outer loop that causes the block-by-block iteration to repeat a given number of times. The number of epochs is hard coded to a value of 3 to ensure the execution time is not too lengthy. Results and analysis are presented in Section 5.8.

4.4.8 Experiment 8: Short-circuiting the Energy Fitness Function. Objective: The purpose of this experiment is to determine the effects of incorporating a short-circuiting operator in the energy fitness function. Both the effectiveness and efficiency aspects are emphasized. The experiment tests both single and multiple node implementations.

Methodology: The algorithm is modified such that it does not compute the remaining sub-functions of the CHARMm energy fitness (Equation 3.12) if a specific value (refer to Table 4.4) is surpassed. The test is set up to run only the Generate Competitive Template portion of the code by setting the Generational inputs to 0. Additionally, the Sweep and Number of Competitive Templates are set to 1 and 50 respectively. The tests are conducted on a single 1GHz machine. Results and analysis are presented in Section 5.9.

4.4.9 Experiment 9: Effects of Ramachandran Constraints. Objective: The purpose of this experiment is to determine the effects of incorporating the Ramachandran Constraints developed by Deerman [25] into the fmGA. Both efficiency and effectiveness are studied.

Methodology: Deerman [25] provides the necessary code that must be integrated into the fmGA, as well as, identifying the code that must be removed. Those changes were

made to the code and a compiler flag was used to indicate when the Ramachandran code should and shouldn't be compiled into the executable program. Unfortunately Deerman only calculated Ramachandran Constraints for the [Met]-Enkephalin and although it would have been possible to calculate new constraints for the Polyalanine₁₄ protein, it was decided that a generalized method needs to be developed so that the algorithm automatically takes into account Ramachandran constraints no matter what the protein under study. A single processor is used with the Ramachandran constraints and without those constraints. Results and analysis are presented in Section 5.10.

4.5 Summary

The methodology outlined in this chapter is used to analyze the various performance changes to the fmGA and its parallelized version (pfmGA). The objective and parameters for each of the nine experiments are laid out as well as the basis for validating the results. The setting up of specific tests to observe particular metrics is fundamental to any research. The recording and analyzing of the results are just as important and is the subject of the next chapter.

V. Results and Analysis

5.1 Introduction

This chapter contains the results and analysis for each experiment discussed in the previous chapter. As previously indicated, each test was run 10 times to ensure *good* statistical results. Conclusions drawn are based solely on the data output of each experiment. Opinions stated that are not based on the actual analysis of the data are so indicated. Raw data is available in electronic format. Visualization of the data is used to help gain a better understanding of the data and relationship between the various input parameters.

This chapter is outlined such that the order of experiments discussed in the previous chapter follows the exact same sequence in this one. Within each experiment, we first present the data obtained. The data is presented in a number of formats (tables, figures, etc.). Upon completion of the data presentation, a subsection dealing with the analysis of the data is presented. It is here that we draw conclusions based upon the data presented.

5.2 Experiment 1: Parallel vs Serial fmGA

Serial fmGA

The fmGA-serialized version of the software was run on a single Sparc Ultra 1 (Sun) Workstation. Although this was not the fastest computer system, it was chosen anyway as the intent of this test run was to obtain fitness data on a serial implementation of the fmGA and timing was not an issue. Note: for each population set, there were 144 generations (iterations) in which the populations were evaluated against the fitness function.

5.2.1 Results and Analysis: Serial. Figure 5.1 shows the results of the 7 independent runs (10 experiments each) where the population sizes were varied from 10 to 500 population members. The figure provides the high-low and mean values obtained. Table 5.1 provides the overall statistical results from the runs sorted by population size. These tests were conducted on the original executable code that was developed by Gates and is assumed to be correct (see Chapter 1 for a list of assumptions).

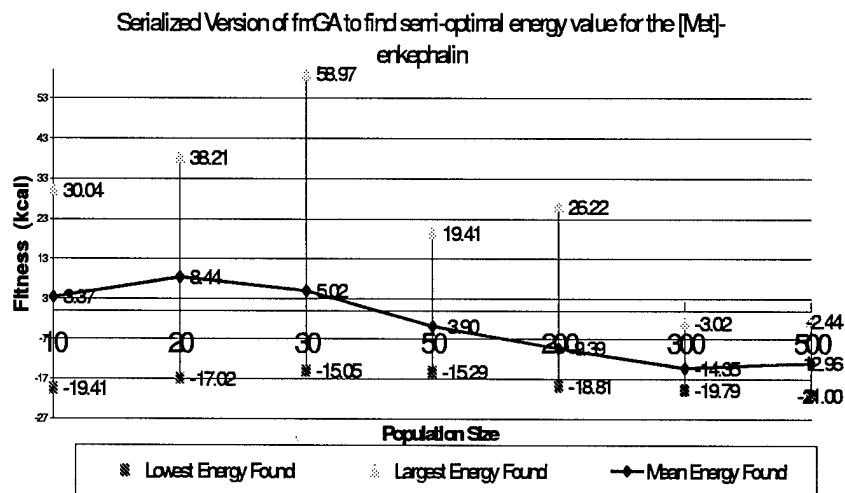


Figure 5.1. Serial Run with Different Population Sizes

Table 5.1. Serial Run with Varying Population Sizes

Population Size	Mean Fitness	Variance	STD	Maximum (Worst) Fitness	Minimum (Best) Fitness
10	3.371	267.813	16.365	30.0418	-19.409
20	8.440	336.924	18.355	38.207	-17.023
30	5.022	499.785	22.356	58.973	-15.051
50	-3.899	126.710	11.257	19.405	-15.287
200	-9.390	189.628	13.771	26.216	-18.811
300	-14.348	25.951	5.094	-3.023	-19.794
500	-12.960	31.953	5.653	-2.436	-21.003

In general, the PSP problem is looking for the minimum energy value and Figure 5.1 shows the algorithm in general finds a lower minimum value as the population size is increased. One other thing worth noting is the fact that the standard deviation decreased as the population size increased. This is expected as the number of fitness evaluations increased with a general tendency for better results to be obtained during each experiment. In other words, because we conduct more searches, we improve the best solution's fitness by a given amount, but we simultaneously improve the worst solution's fitness by an even greater amount, resulting in a shrinking of the standard deviation between the averages as the population size increases.

Parallel fmGA

The parallel fmGA code that was obtained was written for a Paragon Parallel system [41, 93]. The software uses a communications package known as NX. In order to implement the fmGA on the Sun and/or Linux workstations, a software translation of all the communication-hook calls was required (see Chapter 3 and Appendix A for additional details). The converted software was run on four computing platforms; the SP2 and SP3 supercomputers, the pile of PCs, and the Cluster of Workstations. For a detailed description of each platform, please refer to the Appendix B.

To establish a baseline and determine the overall performance of the software, the modified code was executed 10 times on 1, 2, 4, 8, 16, 20, 32, 64, and 96 nodes or as otherwise noted and the population size was set at approximately 200 (refer to Appendix F for further details concerning population sizing).

For all the parallel run results, both time of execution and energy minimization values were kept for each block (blocks 6 thru 10), in each of the 10 experiments for each 1, 2, 4, 8, 16, 32, 64, 96 run. In addition the mean, standard deviation, as well as high/low values were all recorded. Finally, an attempt was made to incorporate high/low or error bars into each of the graphs used to depict the results; however, in the case of the timing data, the variance was too insignificant to be of much value and for the size of the pictures was indiscernable. For the energy minimization value, the fact that the variance was

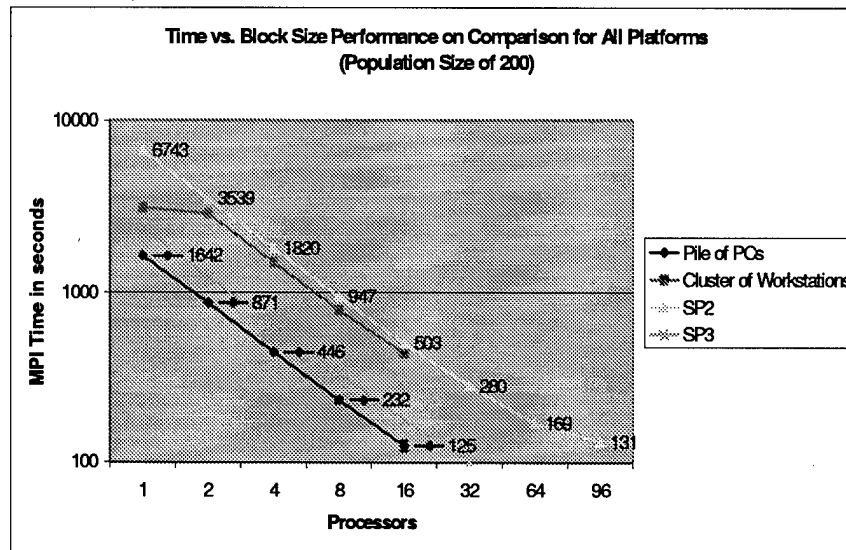


Figure 5.2. Logarithmic Performance Comparison of the Four Computing Platforms

large (compared to the actual values) essentially meant that there was overlap between all results.

An additional run on the SP2 with population sizes of approximately 800 and 4500 was conducted because it was observed that with 8 or more processors, very little improvement on the energy fitness minimization value was seen. In addition, the population size was modified such that both timing and fitness evaluation results could be used as a baseline for future experiments.

5.2.1.1 Results and Analysis: Parallel. The tables and figures in this section depict various results obtained with respect to specific data that the program generated. Figure 5.2 and Table 5.2 provide timing data for all four platforms in both a visual representation and that of a table.

A quick look at the figure reveals a few interesting perks; first, the figure shows that there is a small decrease in timing during the 1 and 2 processor runs but that difference increased from that point on similarly to the other three platforms' results. This occurred due to the fact that the first processor used was faster than the second processor and thus the splitting of the workload taken in conjunction with a slower processor resulted in very little improvement in performance. Recall the Cluster of Workstations, like the Pile

Table 5.2. Mean Completion Times vs Node Size

# of Nodes	Pile of PCs	COW	SP2	SP3
1	1642	3133	6743	2395
2	871	2894	3539	1226
4	446	1503	1819	648
8	231	787	946	331
16	125	441	503	176
32	-	-	280	100
64	-	-	168	-
96	-	-	131	-

of PCs, is a heterogeneous platform having systems with clock speeds between 170MHz to 450MHz. Secondly, without taking into consideration the first run for all platforms, from the 2-processor run to the 64-processor run (depending on the platform) all four platforms showed a similar decrease in runtime. The only caveat is with the SP2, 96-processor run where we notice the speedup between the 64 and 96-processor runs dropped off dramatically. This is due to the fact that there was an insufficient amount of workload to be distributed amongst the 96 nodes and as a result the serial portion of the code (the Generating of the competitive template and all initialization routines) become the limiting factor. It is our belief that if each of the other three platforms were capable of supporting 96 processors then similar results would have occurred and with those platforms that had faster processing capability, that same characteristic would have occurred sooner than when it occurred on the SP2.

Waiting for a program to finish execution is a major factor in determining the software's performance. For this specific implementation, the workload is distributed amongst the various processors (by factoring the population size for each node by the total number of processors used) and we expected to get a reduction in the amount of time it takes to complete as the number of processors is increased. Both Figure 5.2 and Table 5.2 demonstrates the validity of this assumption and shows that for the most part, the algorithm is highly scalable. The figure also shows the overall performance of each platform, with the Pile of PC's performing the best, then the SP3, the cluster of Workstations, and finally the SP2 supercomputer. One way of discussing performance of an algorithm is its ability to

Table 5.3. Speedup Per Number of Nodes

Number of Nodes	Pile of PCs	COW	SP2	SP3
1	-	-	-	-
2	1.88	1.08	1.91	1.95
4	3.68	2.08	3.71	3.70
8	7.11	3.98	7.13	7.24
16	13.14	7.10	13.41	13.61
32	-	-	24.08	23.95
64	-	-	40.14	-
96	-	-	51.47	-

be scaled to a higher number of processors. To demonstrate this metric one merely takes the associated serial time of a given algorithm and divides by the associated time to run the same algorithm on n-nodes. This value is called the Speedup Factor (see Equation 5.1) [78].

$$S = \frac{T_s}{T_p} \quad (5.1)$$

Where S = Speedup and T_s and T_p are Serial and Parallel run times.

Table 5.3 demonstrates the speedup of the parallel fmGA (pfmGA), based on Equation (5.1) between a single node and the specified number of nodes and Figure 5.3 visualizes the results for all four platforms.

Table 5.3 clearly shows a drastic difference in the initial speedup between the 1 and 2-processor runs on the Cluster of Workstations. This occurred as Bucephalus, a 400 MHz 4-node processor computing system was used for the serial run and then a slower machine was used for each subsequent set of parallel runs (SPARC Ultra 170MHz). This also explains why the curve contains the same slope in Figure 5.3 like the other curves with respect to the 2, 4, 8, and 16 nodes from all the other platform runs.

Each of the four platforms appeared to demonstrate the software's high scalability factor. As seen in Figure 5.2, the time to complete the task appears to be linear as the number of processors used to tackle the job is doubled. In fact Table 5.3 shows that

for the most part doubling of the number of processors gives around a 1.8 to 1.9 times the previous speedup in performance. Of course, this tapers off considerably once the serial portion of the algorithm becomes the limiting factor with respect to performance as previously discussed. For the most part, as observed in the previous figures, as the nodes doubled the time to complete nearly halved. In fact, Table 5.2 shows that across the board there is almost a 90% decrease in time as the number of processors/nodes doubles. As there are numerous factors that can affect the amount of time needed to complete each test, the slight differences in speedup between each doubling is understandable.

Finally, as the number of nodes increased to 16 and then 32, we notice that the speedup begins to drop off to the mid 80s and below. The most sound reason for this drop is the fact that as more processors are applied to tackle this particular instantiation (200 population size being the main factor here), there is a theoretical minimum amount of time needed to conduct the work. For instance with 4 processors, we have each processor doing work on 50 members of the total 200 population; for 8 nodes, 25 members; 16 nodes, 13; for 32 nodes, 7 members, etc. And, the way the algorithm is structured, there is a given amount of work that is always done (set up, building the competitive template, etc) that takes "x" amount of time. This work could be considered serial in nature (whether or not it is repeated is not the issue, the fact that it cannot be parallelized is what effects speedup). As a result, as less work is done per node with respect to population members, the serial computational time becomes more and more a factor. To obtain better speedup factors with a larger number of processors, the population size must be scaled up along with the increase in processors.

Table 5.4 provides another comparison of the speedup on the SP2 between a run with the population set at 4500 and the original run with a population set at 200. Unfortunately, the amount of time to complete an entire run on a single processor, with an overall population size of 4500, took more time than was estimated when submitting the job in the batch queue (> 120 hours). As a matter of fact, it wasn't until 4 processors were used that the job could be completed within the 120 total computing hours allotted by the MSRC. The single node run would have taken 375 total computing hours and the 2-node run would have taken 192.6 hours of computing time, thus the runs from 1 and 2

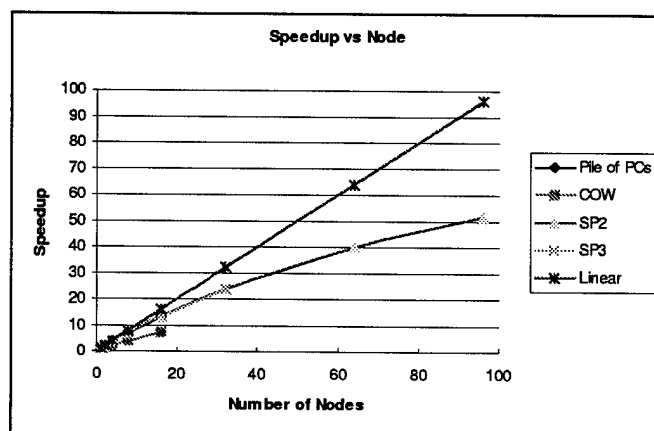


Figure 5.3. Speedup Per Number of Nodes

Table 5.4. Speedup Per Number of Nodes

Number of Nodes	Time to Complete	Speedup (4500)	Speedup (200)
1	135000	1	1
2	69349	1.94	1.91
4	35167	3.84	3.71
8	17961	7.52	7.13
16	9251	14.59	13.41
32	4909	27.51	24.08
64	2707	49.87	40.14
96	1719	78.53	51.47

processors were not conducted 10 times [59]. Notwithstanding this fact, the numbers derived do provide an indication of the overall performance with respect to how much better the speedup is for the fmGA if the population size (workload) is sufficiently increased to a level that allows the parallel workload to dominate the serial workload of the algorithm.

For production purposes, all runs except that of the highest number of nodes is not that important anyway. In fact, so long as the isoefficiency—where the increase in number of processors incorporates an increase in overall workload, such that previous and subsequent runs with these modifications have the same overall efficiency [78]—is good, given a sufficient amount of work to be done by each processor/node and if the time it takes to complete the entire run is satisfactory, then the ability of the software to run to completion within the same time frame but with fewer nodes attacking the problem is of no concern.

With this in mind, we suspect the pfmGA software is scalable to even higher numbers of nodes, so long as sufficient workload is provided and especially when we consider that the amount of information passed between processors is less than 5KBytes for a single solution to be passed per processor (see Equation 5.2). Using the equation, we calculate that 2.808KBytes of information are transferred for a single Polyalinine₁₄ solution. In a parallel environment this amount of information would be sent n times; where n is the total number of nodes in the parallel configuration minus one. In addition, it appears that the best results, with respect to energy minimization, is when we have larger-sized populations per node. Of course, as that number increases, the overall time to completion also increases. For production purposes, one would need to set a time length for the job to run, then adjust the population size such that the job would complete within the allotted time. Finally, Figure 5.4 shows the associated times it took to complete various stages of the actual pfmGA PSP software. As expected, those runs with smaller population sizes generally took less time to complete, although if we had executed the software with more than 96 processors, then a run having a population size of 4500 might have finished before that of a run with a population size of 800 having only 8 processors to tackle it. Note: Because this is logarithmically scaled, error plots of the runtimes were not included as they would have been insignificant and would not have shown well.

$$Bytes = Num-S-Sent*((S_{Str-Size}*Sizeof(Char))+(S_{fitness})+(S_{Str-Size}*Sizeof(Int))) \quad (5.2)$$

where Num-S-Sent = number of solutions sent is 1

$S_{Str-Size}$ = string size is 240, 560 for [Met]-Enkephalin and Polyalanine₁₄ respectively

Sizeof(Char) = 1 Byte

Sizeof(Int) = 4 Bytes

$S_{fitness}$ = 8 Bytes

The important thing here is not the above comparison, but the fact that the time difference from block to block increases ever so slightly. This occurs because the population size is directly related to the current building block size as the following numbers demonstrate. For 6, 7, 8, 9, 10 block sizes with the *n_a* parameter set at 4500 on two nodes we have 2619, 2768, 2950, 3173, 3443 population sizes per each node. To find out more about the derivation of population size, please refer to Appendix F. A quick look at the numbers shows that for the 2 processor, 4500 population size run, subsequent increases in time between blocks 6 and 7, 7 and 8, 8 and 9, 9 and 10 were 13087, 13958, 15005, and 16328, respectively. The increase in population size due to the size of the block would seem to justify the slight increases in subsequent block times.

Table 5.5 clearly shows that as the number of processors increase, with the *n_a* parameter held constant, results in the overall effectiveness of the algorithm decreased. Recall that the *n_a* parameter directly attributes to the overall population size, which of course is subdivided amongst the nodes resulting in lower population sizes per node as the number of nodes is increased. This result is clearly what one would expect as the amount of work done by each node becomes less and less (less population members to work on). Figure 5.5 shows how many evaluations were done per processor based on *n_a* in all 10 experiments. As the *n_a* value increases, so did the calls to the fitness evaluation function per node for a given # of parallel nodes used; however, as the number of nodes was increased, the fitness evaluation calls per node decreases. The figure also shows that

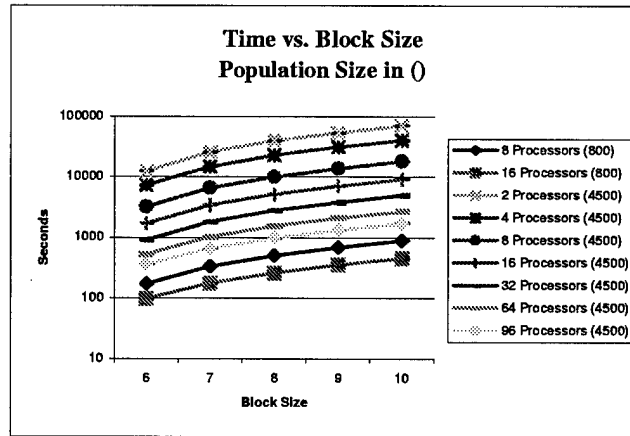


Figure 5.4. Mean Time to Complete Each Building Block

Table 5.5. Fitness Evaluation Results Ordered by the n_a Parameter

Number of Nodes	Best Fitness	Mean Fitness	Std Dev	n_a Parameter
1	-27.668	-26.246	1.118	200
2	-27.721	-25.083	1.282	200
4	-26.323	-24.045	1.411	200
8	-25.995	-23.686	1.533	200
16	-25.995	-23.664	1.556	200
32	-25.995	-23.691	1.528	200
64	-25.995	-23.694	1.532	200
96	-25.995	-23.671	1.549	200
8	-27.721	-25.083	1.282	800
16	-26.323	-24.045	1.411	800
1	-29.103	-27.764	1.034	4500
2	-28.851	-27.289	1.194	4500
4	-28.067	-26.071	1.996	4500
8	-28.263	-25.829	2.434	4500
16	-27.399	-25.681	1.238	4500
32	-28.423	-26.436	1.001	4500
64	-26.013	-24.344	1.157	4500
96	-26.028	-24.035	1.408	4500

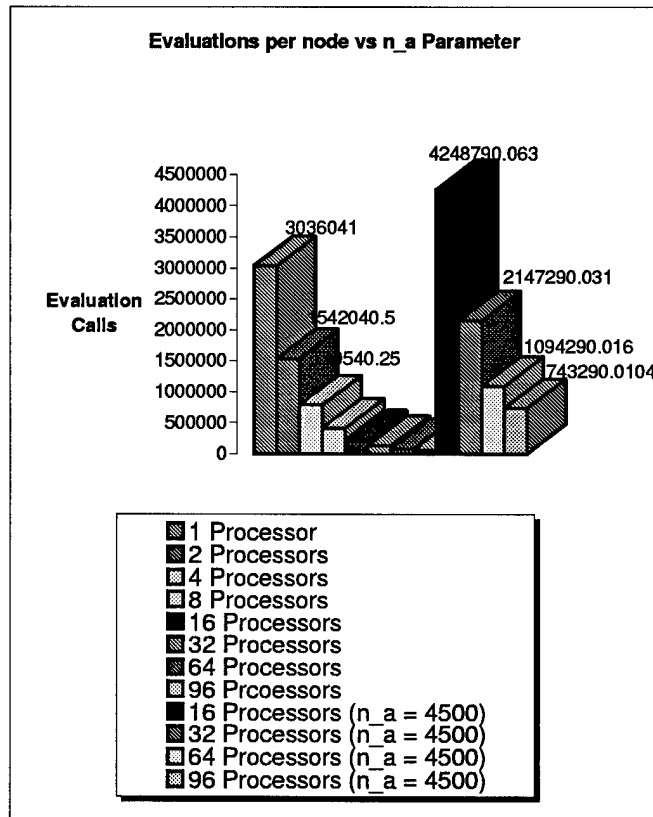


Figure 5.5. Fitness Evaluation Calls per Number of Nodes

the better fitness evaluations occur where there were more calls to the fitness evaluation function per node (i.e. $n_a = 200$, 1 node, $n_a = 4500$, 16 nodes)

Finally, Table 5.6 is merely a rearrangement of Table 5.5 to allow the reader to clearly see that as the population size is increased (by increasing the n_a function) and the number of nodes is held constant, then the best fitness found also increases.

Figure 5.6 shows how the worst and best fitness evaluation results overlap for all cases with $n_a = 200$ (population size of approximately 200). An interpretation of this could mean that statistically speaking, there is no distinction between the overall results no matter how many processors are used to attack the problem.

Finally, Figure 5.8 shows what happened to the standard deviation with respect to time as the number of processors is increased. The first thing to notice is the fact that as the number of processors increased, there was a tendency for the standard deviation to

Table 5.6. Fitness Evaluation Results Ordered by Number of Nodes

Number of Nodes	Best Fitness	Mean Fitness	Std Dev	n_a Parameter
1	-27.668	-26.246	1.118	200
1	-29.103	-27.764	1.034	4500
2	-27.721	-25.083	1.282	200
2	-28.851	-27.289	1.194	4500
2	-28.397	-26.359	1.127	4600
4	-26.323	-24.045	1.411	200
4	-28.067	-26.071	1.996	4500
8	-25.995	-23.686	1.533	200
8	-27.721	-25.083	1.282	800
8	-28.263	-25.829	2.434	4500
16	-25.995	-23.664	1.556	200
16	-26.323	-24.045	1.411	800
16	-27.399	-25.681	1.238	4500
32	-25.995	-23.691	1.528	200
32	-28.423	-26.436	1.001	4500
64	-25.995	-23.694	1.532	200
64	-26.013	-24.344	1.157	4500
96	-25.995	-23.671	1.549	200
96	-26.028	-24.035	1.408	4500

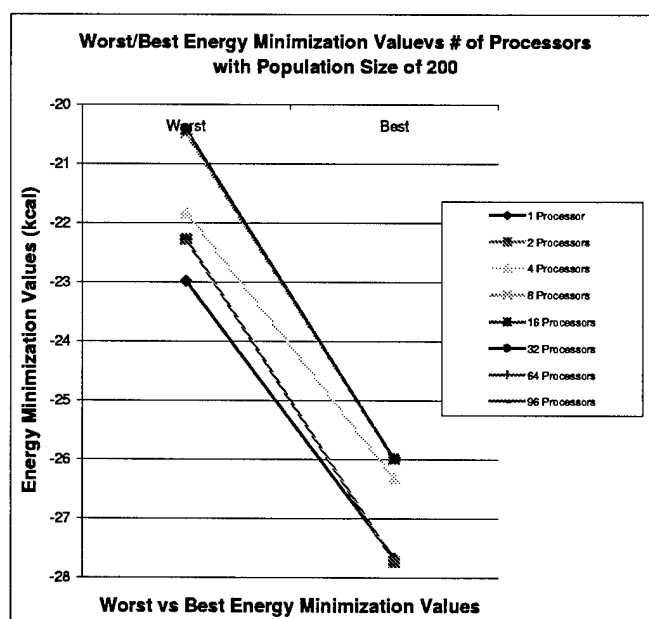


Figure 5.6. Worst/Best Fitness Evaluation Results

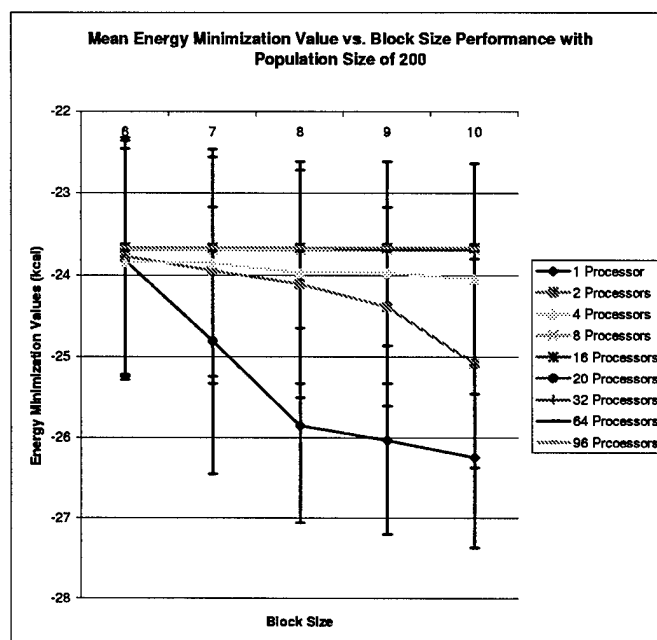


Figure 5.7. Fitness vs Building Block Size

decrease. The amount of the standard deviation, with respect to the overall mean, wasn't very significant and was one of the reasons why they weren't included in all the figures. Recall from previous discussion that it was noticed that the COW runs experienced some NFS delays and are probably behind the reason for the extreme difference. Finally, notice how quickly the standard deviation was reduced to less than 1 on the faster platforms (Pile of PCs and SP3) and eventually getting there on the slower two platforms after there had been a significant reduction in the population size per node (by increasing the number of nodes used to tackle the problem). This observation leads us to think that the speed of the processor is a large factor in determining how much of a standard deviation (faster processor means a smaller standard deviation) to expect and/or the number of processors used during a run (more signifying a smaller standard deviation). Note: Although we did not attempt to normalize the standard deviation, we suspect that a comparison of normalized standard deviations would show that the standard deviations across the platforms are nearly identical.

The overall results of the serial and parallel versions of the fmGA were what one would expect when applied to a very complex problem such as the protein structure prediction

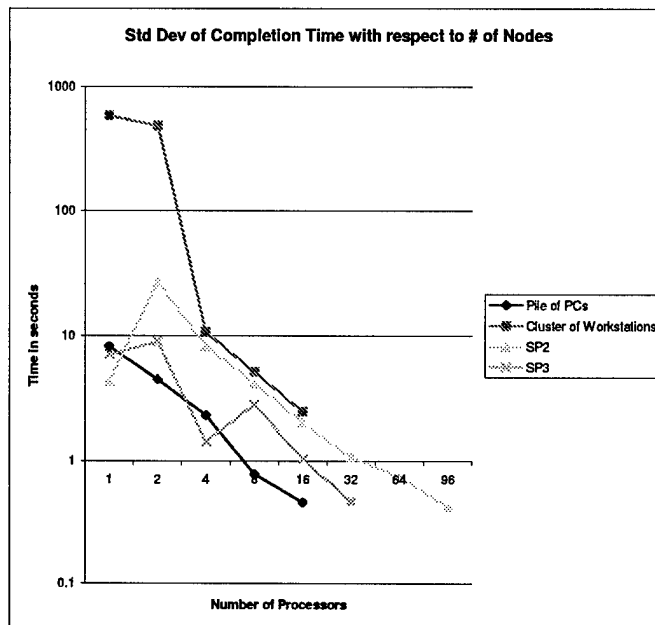


Figure 5.8. Standard Deviation of Runtime for Different Computing Platforms

problem. There was no guarantee that by changing the population size we would get a more “fit” result; however, the trend did seem to indicate that, in general, the more evaluations you perform, the better the answer you “should” find.

The overall goal of this series of tests for the first experiment was to establish a baseline which can be used to compare future enhancements (next series of experiments) of the software. It was assumed that the code used was correctly designed and was the latest version available in AFIT's GA Toolkit. In addition, the intent of this set of tests was to examine the underlying performance of the fmGA with respect to only a single parallel implementation (there are many ways the software could be implemented in parallel and the reader is encouraged to review Gates and Merkle's paper where the parallel fmGA is executed a number of times under different parallel configurations [93]).

5.3 Experiment 2: Generating Competitive Templates with a Hill-Climbing Sweep Function

5.3.1 Results. The set of experiments completed here imply that there are two competing dynamics at play when trying to build a good competitive template. Figure

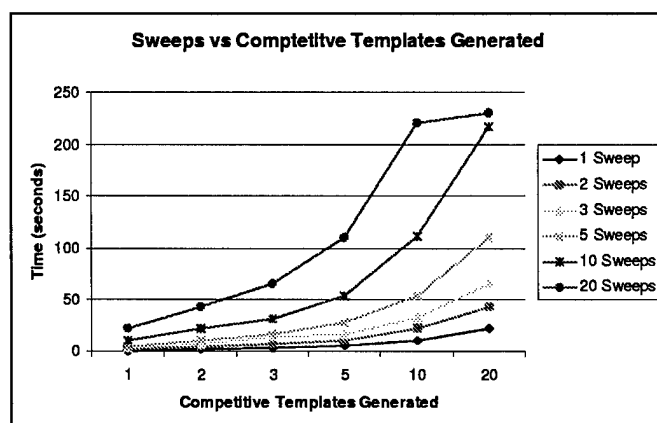


Figure 5.9. Generating Competitive Template vs Sweep (Timing)

5.9 shows the length of time required by a single processor (600MHz) to generate a given number of competitive templates with a specific number of sweeps. Recall that the sweep function was merely a local, hill-climbing technique where a single bit is flipped, the results evaluated (and kept if better) with the process continuing until all bits have been attempted to be flipped. The figure also shows that as the number of competitive templates is increased, the time to generate them also increases, but as the number of sweeps is increased it was only until the 15th sweep that the modification to the code (where we terminate the sweeping function if no better solution is found) took affect. If the termination function had not been added we would have seen the time it takes to sweep 20 competitive templates 15 times to be twice as long as sweeping 10 competitive templates 15 times.

It is interesting to note that prior to this experiment, a series of competitive templates were generated with unlimited sweeps to determine the average number of sweeps it takes before no improved competitive template is created. For the Met-Enkephalin the average was 7.5 sweeps and for the Polyalanine protein it was nearly 13 sweeps. This leads us to suspect that the larger the protein the more sweeps would be required before a local minima can be obtained; however, there is no guarantee that this would hold and thus requires further study.

Figure 5.10 demonstrates the affect of generating a different number of competitive templates for a specified number of sweeps on the fitness of the best competitive template found.

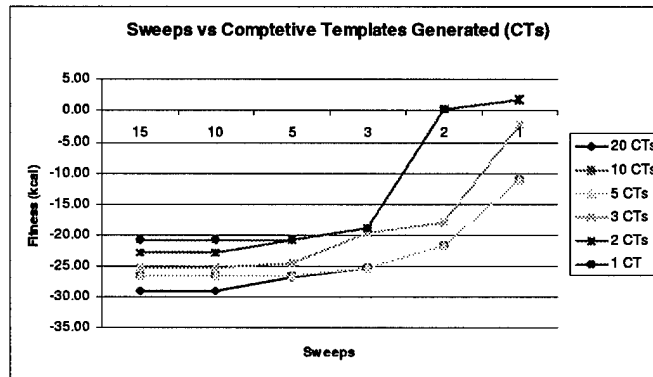


Figure 5.10. Generating Competitive Template vs Sweep (Fitness)

5.3.2 Analysis. The intent of this set of simple tests was to determine the cost vs. benefit of generating good competitive templates via the sweeping function. The standard deviation was not plotted on the graph as they overlapped significantly for all cases and trying to determine where one began and the other one ends would be difficult. It is evident that the greater number of competitive templates found with the highest number of sweeps required to find no better solution results in the best possible competitive template to be generated stochastically. Finally the new terminating code for the sweep function worked as shown earlier. In the case where the amount of time it took to sweep 15 times while generating 20 competitive templates, it doesn't take much longer than sweeping through 15 times while generating 10 competitive templates. Even though this terminating code worked and the results were good with respect to both time and effectiveness we decided upon only using three and five for the sweeps and competitive templates input variables. This decision was made mostly due to the fact that the timing costs associated with the increased values for the sweeps and competitive templates was deemed too costly.

5.4 Experiment 3: Random Search with and without the Hill-Climbing Sweep Function

5.4.1 Results. There were two tests that were run during this experiment. The first test was to merely generate a large number of strings, locally optimize them with the Hill-Climbing Sweep method, then evaluate them. The second test was to generate a large number of random strings and evaluate them. The results are then compared to a serial fmGA run of similar cost in time.

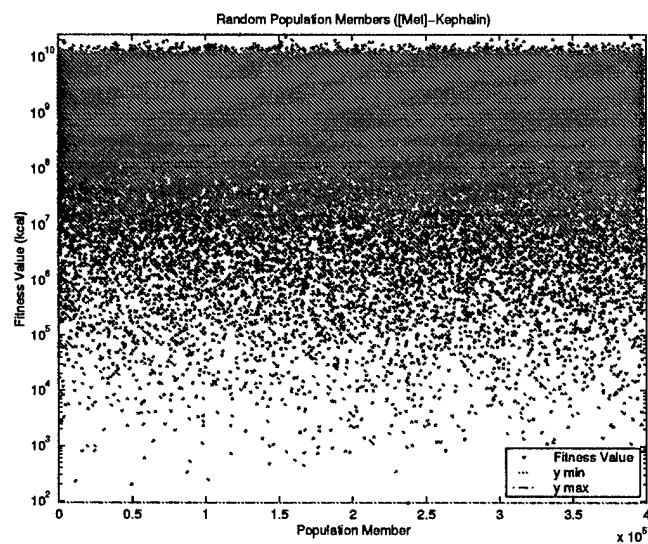


Figure 5.11. Randomly Generated Points vs Fitness

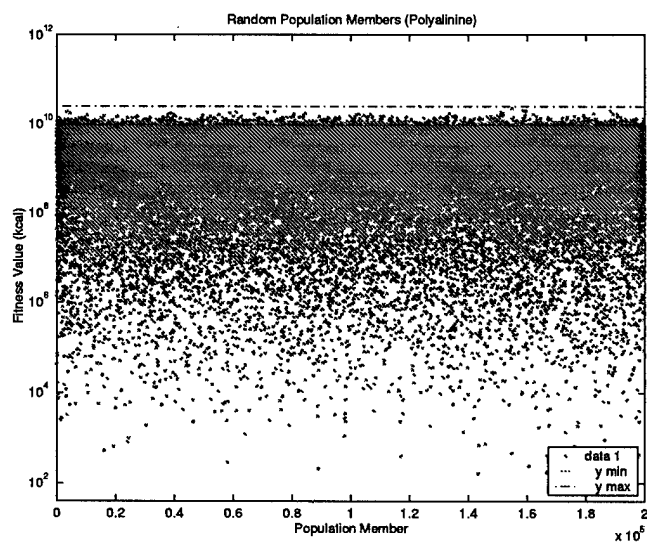


Figure 5.12. Randomly Generated Points vs Fitness

Table 5.7. Energy (kcal) Fitness Values for Random Search With and Without Sweeps

Random (No Sweeps)	Met-Enkephalin Values (kcal/mol)	Polyalinine ₁₄ Values (kcal/mol)
Max	2.4742×10^{10}	2.474×10^{10}
Min	93.5980	88.94
Average	2.6581×10^9	2.65×10^9
Std Dev	2.1875×10^9	2.176×10^9
Median	2.1754×10^9	2.17×10^9
Random (With Sweeps)		
Max	-17.983	-171.274
Min	-28.313	-332.867
Average	-22.139	-309.218
Std Dev	7.391	18.998
Median	-21.358	-312.882
Serial fmGA		
Max	-22.2764	-314.505
Min	-27.7212	-328.341
Average	-26.246	-322.147
Std Dev	1.282	4.874
Median	-26.414	-323.684

5.4.2 Analysis. Table 5.7 provides us an overview of how well both random tests fared with respect to a serial run of the fmGA (original code). Though timing information is not shown, the sweeping mechanism was extremely costly in that the other algorithms (Random without Sweeping and the serial version of the fmGA) took less than 1500 seconds while the Random with Sweeping algorithm took nearly 3000 seconds (plus or minus 125.8 seconds) to generate only 25 random population members (it took over 30,000 seconds (plus or minus 323.4 seconds) to generate 250 “swept” members). Although in both cases the random sweep found a better solution, it had difficulty repeating that feat as shown by the average and median statistics. These results tend to confirm our belief that a random search without any other additional work is not a worthwhile endeavor. While the random search with the sweep did okay and found a better answer for the set of tests conducted in this experiment, it is still out performed by the fmGA in other experiments. If you recall, Figure 2.6 sort of demonstrated the fact that there are a lot of points that have extremely high fitness values and finding those points that are good is very difficult.

Table 5.8. Stochastic Search Speedup

Processor Speed (MHz)	Speedup Percentages			
	Average	Max	Median	Min
333	0.998	1.059	0.999	0.923
600 (Server)	1.770	1.859	1.774	1.633
600	1.796	1.922	1.797	1.654
600	1.806	1.900	1.809	1.678
600	1.807	1.930	1.809	1.661
600	1.818	1.903	1.821	1.661
600	1.823	1.969	1.824	1.682
1000	2.827	2.994	2.836	2.616
1000	2.839	3.045	2.842	2.630
1000	2.863	3.041	2.858	2.693
1000	2.864	3.069	2.865	2.636
1000	2.865	3.014	2.869	2.644

5.5 Experiment 4: Improving Performance in a Heterogeneous Environment

5.5.1 Results. Table 5.8 presents the speedup that was realized for the 12 machines over 10 runs in our heterogeneous environment. The Stochastic Search Speedup (*SSS*) is defined by Equation (5.3) to be the percentage of time that the faster machines complete a specific section of the algorithm as compared to the slowest machines completion time over the same section of the algorithm.

$$SSS = \frac{\text{SlowestMachine'sBlockExecutionTime}}{\text{CurrentMachine'sBlockExecutionTime}} \quad (5.3)$$

Each of the 600 MHz machines achieve, on average, the theoretical best *SSS* of 1.714 over the 350 MHz machine. The same holds true for the 1 GHz machines which achieves, on average, the theoretical best *SSS* of 2.857 over the 350 MHz machine.

The utilization of available computational time, for each population size, is presented in Table 5.9. The fitness calls are the total number of fitness calls that were realized in the two versions of the load balanced pfmGA. The number of additional fitness calls that were made in the load balanced pfmGA as compared to the original pfmGA are represented by the additional fitness call values. Since the same exact machines were included for both the original pfmGA and the load balanced tests, the time utilized is identical for both. This time is totally dependent upon the slowest machine utilized, which is 333 MHz in our testing.

Table 5.9. Utilization of Available Computational Time

Load Balanced pfmGA					
Population Size	Fitness Calls	Additional Fitness Calls	Time Utilized (Hours)	Time Available (Hours)	Utilization Percentage
200	21,104,924	10,659,371	26.871	12.034	0.975
400	41,877,670	21,142,131	52.417	24.274	0.988
800	81,386,551	41,318,879	101.978	47.856	0.982
1200	121,889,301	61,289,999	151.422	73.049	0.996
1600	162,139,398	81,983,466	198.411	90.172	0.991
Load Balanced pfmGA Baldwinian Approach					
Population Size	Fitness Calls	Additional Fitness Calls	Time Utilized (Hours)	Time Available (Hours)	Utilization Percentage
200	20,978,381	10,532,828	26.766	12.216	0.976
400	41,569,035	20,833,496	52.208	24.002	0.988
800	81,298,313	41,230,641	101.804	48.035	0.994
1200	120,384,245	59,784,943	149.464	70.486	0.996
1600	161,938,214	81,782,282	193.458	81.218	0.997

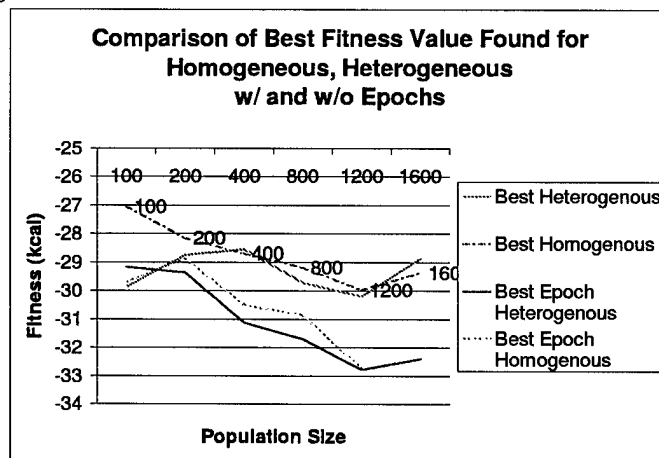
The utilization percentage, presented in Table 5.9 is the percentage of the time available that the faster machines were actually able to utilize, in the load balanced case, over the 10 runs. The time available represents the total amount of time that any of the machines were sitting idle in the original pfmGA and may now be utilized in the load balanced case. The time available is calculated by subtracting the time that the current machine took to execute a piece of the code from the time the slowest machine took for the same portion of the code in the previous iteration. This is a good approximation of the amount of time that is available for computation in the load balanced case. During this time available, the faster machines conducted the appropriate number of extra searches on the competitive template and other population members. This was accomplished by determining the amount of time, dynamically, for each search. This search time is used in conjunction with the time that is available to determine how many extra searches can be conducted. It is also noted that as the population size is increased, the time available is increased and the utilization percentage achieved also increases. This illustrates that for larger workload sizes, our dynamic load balancing mechanism's efficiency increases.

The resulting values are presented to the user as solutions to the energy minimization problem in Table 5.10. The actual fitness values in terms of average, min, max, and median that were obtained using the original pfmGA and the Load Balanced pfmGA over 10 runs are presented.

Table 5.10. Protein Energy (kcal) Fitness Values

Original pfmGA					
Population Size	Maximum Fitness	Median Fitness	Minimum (best) Fitness	Average Fitness	Standard Deviation
200	-25.988	-27.083	-28.280	-27.106	0.737
400	-26.908	-28.480	-32.458	-29.003	1.932
800	-26.032	-29.854	-32.955	-29.391	2.092
1200	-29.397	-29.874	-31.176	-30.137	0.609
1600	-28.441	-29.172	-30.953	-29.412	0.780
Load Balanced pfmGA					
Population Size	Maximum Fitness	Median Fitness	Minimum (best) Fitness	Average Fitness	Standard Deviation
200	-24.067	-28.134	-31.355	-27.119	1.577
400	-24.036	-28.563	-31.534	-28.728	1.695
800	-25.211	-28.685	-32.219	-29.170	1.810
1200	-25.138	-30.536	-33.666	-30.176	1.977
1600	-25.069	-30.286	-33.743	-30.501	1.477
Load Balanced pfmGA With Baldwinian					
Population Size	Maximum Fitness	Median Fitness	Minimum (best) Fitness	Average Fitness	Standard Deviation
200	-25.949	-27.484	-30.229	-27.771	1.308
400	-27.346	-28.930	-30.958	-28.761	1.045
800	-28.373	-29.344	-34.967	-30.049	2.104
1200	-29.508	-30.252	-32.527	-30.482	1.075
1600	-28.229	-30.244	-31.012	-29.837	1.083

Figure 5.13. Comparison Between Homogeneous and Heterogeneous (Best) Fitness Values



5.5.2 Analysis. The most astounding aspect of this work is the realization of the -34.967 energy value for the pentapeptide [Met]-Enkephalin. This is the lowest energy value ever found using the CHARMM energy model with the pfmGA algorithm. The variety of values is due to the stochastic search process.

From the results, one can clearly see that without load or task balancing a given parallel algorithm, the potential to waste computational speed is readily apparent. Our pfmGA algorithm wastes nearly 50% of the computational ability of our heterogeneous Pile of PCs. The modified code clearly demonstrates the ability to utilize more than 97% of "what would have been" wasted computational time. Additionally, the way the extra time was used to conduct local "baldwinian" searches proved to enhance the fmGA's overall effectiveness! Figure 5.13 clearly shows that for most population sizes, the heterogeneous implementation found better fitness values than the homogeneous implementation of the fmGA. It should be noted that the overall best fitness found for the Met-Enkephalin was found during this test. The angular representation and its RMSD from the optimal value is discussed at the end of this chapter.

5.6 Experiment 5: Searching for a Secondary Structure

5.6.1 Results. The results from the first set of experiments is presented in Table 5.11. These experiments were run to determine what effects modifying the input constraint parameter would have on the overall effectiveness of the algorithm on the peptide model Polyalanine₁₄. Therefore, the algorithm was executed 10 times for different percentage values on a single processor with a constant population size of 30. In each of these runs, both the dihedral angle constraint and the percentage constraint described earlier were kept the same as well as the fmGA parameters previously specified.

5.6.2 Analysis. The results from Table 5.11 indicate that the use of the SSA produced the best results, in terms of the best result found, with the SSA percentage set between 15-90%. What can be concluded from this is that the secondary structure analysis is integral to finding improved solutions! Additionally, the low and high ends did not produce very good results since both extremes require the population to have the

Table 5.11. Secondary Structure Constraint Analysis (in kcal)

fmGA with SSA (model Polyalanine)					
SSA %	Max Fitness	Median Fitness	Best Fitness	Average Fitness	Stand Dev
0%	-100.160	-125.449	-136.433	-123.573	10.881
5%	-110.490	-130.933	-133.811	-125.449	9.635
10%	-101.837	-128.188	-138.137	-123.588	13.425
15%	-110.286	-130.105	-140.560	-129.495	8.943
20%	-104.369	-134.745	-143.786	-131.767	11.054
25%	-120.941	-132.295	-139.384	-131.469	6.233
30%	-116.572	-136.028	-139.445	-132.374	8.725
35%	-107.275	-135.722	-145.900	-133.244	10.406
40%	-106.880	-131.993	-137.386	-128.940	9.262
45%	-98.273	-133.537	-145.450	-131.819	13.072
50%	-104.501	-132.501	-143.801	-130.846	10.594
60%	-123.386	-137.333	-145.439	-136.655	6.564
70%	-122.441	-133.848	-141.089	-133.201	5.813
80%	-109.138	-136.016	-145.695	-133.499	10.880
90%	-125.925	-137.140	-145.923	-136.323	6.777
100%	-93.407	-127.440	-131.671	-122.959	12.089

Table 5.12. Protein Energy (kcal) Fitness Values

fmGA without SSA ([Met]-Enkephelin)					
Pop Size	Max Fitness	Median Fitness	Best Fitness	Average Fitness	Stand Dev
100	-22.189	-26.133	-29.598	-25.976	2.045
200	-22.721	-26.114	-28.075	-26.167	1.606
400	-26.608	-27.582	-30.315	-27.865	1.240
800	-23.979	-27.061	-30.141	-26.899	1.991
fmGA with SSA ([Met]-Enkephelin)					
Pop Size	Max Fitness	Median Fitness	Best Fitness	Average Fitness	Stand Dev
100	-23.860	-25.181	-29.615	-25.675	1.579
200	-23.356	-26.355	-29.389	-26.347	1.739
400	-25.349	-27.122	-30.054	-27.288	1.548
800	-24.973	-27.304	-30.041	-27.593	1.642
fmGA without SSA (model Polyalanine)					
Pop Size	Max Fitness	Median Fitness	Best Fitness	Average Fitness	Stand Dev
100	-107.970	-125.792	-137.711	-126.745	9.766
200	-114.521	-136.491	-140.097	-131.804	8.961
400	-127.158	-136.440	-143.126	-135.559	5.183
800	-137.429	-139.203	-150.731	-140.893	4.377
fmGA with SSA (model Polyalanine)					
Pop Size	Max Fitness	Median Fitness	Best Fitness	Average Fitness	Stand Dev
100	-120.727	-131.821	-146.498	-134.587	7.948
200	-132.302	-138.315	-148.532	-138.840	4.458
400	-134.452	-139.478	-149.222	-140.618	4.432
800	-133.226	-140.827	-152.053	-140.920	4.743

exact angles of the secondary structure or a large percentage of those angles present in the population. There is some minor fluctuation present in the results obtained between 15-90% as would be expected with a stochastic algorithm.

Table 5.12 illustrates that our modified fmGA algorithm is more effective at finding a lower energy value when utilizing the SSA. The data obtained from the SSA modified fmGA when applied to the Polyalanine peptide indicates that in all cases of population sizes tested, the standard deviation, median, best, and average fitness values were lower i.e. improved over the results obtained without the SSA. The results obtained from the fmGA as applied to the [Met]-Enkephalin pentapeptide showed no considerable difference with or without the SSA. This was expected since the [Met]-Enkephalin pentapeptide does not have a secondary structure. Additionally, the use of the SSA has very little overhead in terms of the computational cost! In all experiments run, the number of additional fitness calls conducted with the SSA were less than 0.1% of the number of fitness calls conducted without the SSA. Therefore the SSA essentially does not increase the specific computational requirements of the fmGA algorithm.

It is clear that the Polyalanine peptide which is nearly 300% larger than the Met-Enkephalin, in terms of the number of residue angles and almost 250% larger in terms of the data structure takes a considerable amount of time to analyze; however, the goal of this paper was not to find a solution to larger proteins in the same time as smaller ones but instead to find "good" solutions to the larger proteins. The results presented here validate our work as an improvement in the effectiveness of the fmGA.

5.7 Experiment 6: Effects of Seeding the Population

5.7.1 Results. The experiments were designed to provide enough data to complete a statistical analysis of the results. For each of the population seeding methods of secondary structures and other methods previously identified in Chapter III and percentages (0%, 10%, 20%, 30%, 40%, 50%, and 100%) of seeding presented in Tables 5.13 and 5.14, 10 data runs were completed for statistical significance. The seeding percentages specify the percentage of the number of initial population members which are seeded via one of the aforementioned seeding methods. The maximum, median, minimum and average fitness

Table 5.13. Met-Enkephalin Energy (kcal)

		[Met]-Enkephelin				
	%	Max	Median	Min	Avg	SD
	0	-20.94	-23.90	-26.35	-23.78	1.52
Sweeps	5	-20.94	-23.90	-26.35	-23.78	1.52
	10	-24.97	-26.39	-27.69	-26.35	1.52
	20	-26.06	-26.99	-28.78	-27.21	0.76
	30	-26.08	-27.10	-28.60	-27.18	0.91
	40	-26.71	-27.93	-28.93	-27.92	0.64
	50	-27.88	-27.85	-30.01	-27.96	0.90
Combo	5	-23.75	-25.25	-27.94	-25.47	1.38
	10	-24.60	-26.20	-28.68	-26.50	1.48
	20	-25.16	-26.10	-28.28	-26.44	0.96
	30	-25.76	-26.43	-27.49	-26.46	0.55
	40	-25.10	-26.97	-31.26	-27.17	1.70
	50	-25.92	-27.07	-29.09	-27.29	1.14
α -helix	5	-21.41	-24.34	-27.16	-24.23	1.79
	10	-22.97	-24.67	-27.34	-24.84	1.57
	20	-22.91	-24.52	-26.94	-24.78	1.13
	30	-21.00	-24.37	-28.35	-24.40	1.87
	40	-23.12	-26.08	-29.04	-25.89	1.66
	50	-23.35	-25.48	-29.21	-25.52	1.78
β -sheet	5	-21.60	-24.24	-27.16	-24.19	1.69
	10	-22.87	-25.29	-27.90	-25.21	1.53
	20	-23.37	-25.21	-29.84	-25.61	1.69
	30	-20.96	-25.67	-29.40	-25.47	2.32
	40	-23.77	-25.46	-26.36	-25.23	0.98
	50	-21.24	-25.81	-27.02	-25.48	1.62

values are presented along with the associated standard deviations. The minimum fitness value is also the best fitness found considering the PSP minimization problem. Both protein molecules used have unique properties that should help discriminate the effectiveness of each seeding mechanism. The Met-Enkephalin contains no secondary structure while the Polyalanine has a perfect α -helix secondary structure.

Note: The median values and average values are close together in most cases along with a relatively small standard deviation. Therefore given the max and min values, the distribution of points in the energy landscape imply an energy surface that is somewhat coarse. Thus, for these particular proteins it is somewhat difficult to search with an evolutionary algorithm.

5.7.2 Analysis. The fitness results obtained from each test set are presented in Tables 5.13 and 5.14. When compared to their computational performance, they show

Table 5.14. Polyalanine₁₄ Energy (kcal)

		Polyalanine				
	%	Max	Median	Min	Avg	SD
	0	-111.99	-127.77	-140.56	-128.78	8.55
Sweeps	5	-119.75	-130.84	-130.76	-139.25	4.95
	10	-128.76	-135.15	-136.92	-130.76	3.28
	20	-128.77	-133.79	-137.24	-134.83	3.47
	30	-131.49	-133.79	-141.03	-134.83	3.12
	40	-132.77	-134.64	-136.76	-134.59	1.33
	50	-131.58	-135.67	-139.76	-135.94	2.71
Combo	5	-118.72	-129.21	-136.48	-128.61	5.53
	10	-124.01	-130.91	-142.41	-130.89	5.53
	20	-128.50	-132.15	-139.30	-132.83	3.57
	30	-126.36	-133.14	-137.01	-132.68	3.00
	40	-127.41	-133.12	-139.97	-133.72	4.08
	50	-129.12	-132.20	-144.19	-133.66	4.38
α -helix	5	-118.45	-130.22	-137.34	-129.37	7.05
	10	-116.36	-126.12	-140.59	-127.64	8.64
	20	-116.17	-132.26	-138.57	-129.63	8.50
	30	-116.46	-129.68	-138.95	-129.22	7.98
	40	-121.18	-126.41	-136.07	-127.34	4.93
	50	-117.96	-127.11	-140.08	-127.70	7.81
β -sheet	5	-118.89	-127.56	-137.34	-128.46	6.25
	10	-119.76	-130.95	-141.00	-131.18	6.02
	20	-114.75	-130.81	-138.57	-128.68	8.62
	30	-120.44	-131.61	-136.67	-131.16	4.44
	40	-107.38	-127.84	-138.02	-125.81	9.95
	50	-108.08	-127.50	-142.72	-126.33	9.30

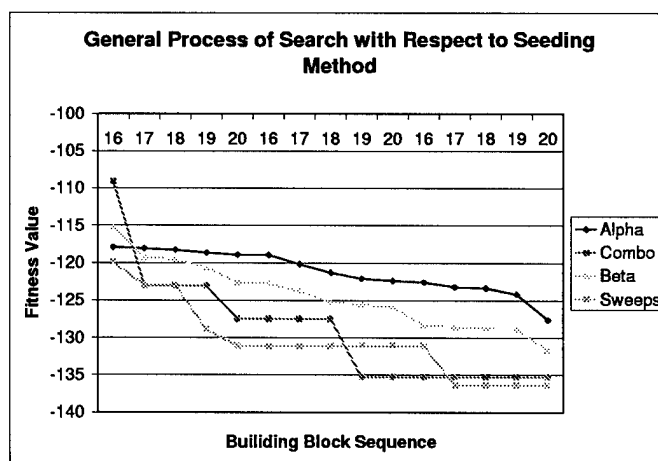


Figure 5.14. Building Block vs. Fitness Progress

Table 5.15. Protein Timing and PVE

		M-Timing		P-Timing			
	%	Avg	SD	PVE	Avg	SD	PVE
	0	894	4.2	37.6	4939	16	38.35
Sweeps	5	864	4.1	37.3	5926	52	45.33
	10	1012	5.9	38.4	7426	52	55.67
	20	1254	4.3	46.1	10351	17	77.67
	30	1446	6.3	53.2	13255	20	98.31
	40	1689	4.8	60.5	16289	23	121.02
	50	1925	5.0	68.9	19359	37	142.41
Combo	5	1351	16.3	53.0	4922	170	38.3
	10	1433	18.6	54.1	5596	211	42.8
	20	1590	26.0	60.1	6551	177	49.3
	30	1675	42.1	63.3	7389	246	55.7
	40	1112	13.9	40.9	8410	190	62.9
	50	1167	25.1	42.8	9577	278	71.7
α -helix	5	889	5.5	37.0	4912	25	38.0
	10	879	4.0	37.4	4919	25	38.5
	20	906	5.7	36.1	4904	25	37.8
	30	901	3.8	35.3	4919	23	38.1
	40	887	3.1	36.8	4905	22	38.5
	50	874	3.7	35.2	4900	22	38.4
β -sheet	5	893	4.9	36.9	4951	39	38.5
	10	896	5.3	35.6	4913	23	36.9
	20	884	3.2	34.5	4907	24	38.1
	30	876	3.5	34.4	4904	16	36.4
	40	876	3.3	34.8	4913	23	39.1
	50	886	5.7	34.8	4907	15	38.8

that while both the HCI and Combo Seeding methods obtained better averages, it wasn't without cost. In fact, Tables 5.13 and 5.14 show that a considerable cost is required to obtain better results.

In order to compare the various methods employed, we combine the two variables (fitness and time) into a new single variable. This new metric is introduced to allow a direct comparison between different runs as a single value. The new metric, Equation (5.4), is calculated from average fitness and timing values while the standard deviations with respect to both of them are ignored. While neither the α nor β SSI methods did not beat a randomly generated population, they did find better solutions at a significantly cheaper cost (see Tables 5.13 and 5.14). A review of the new metric values show that there is a slight advantage in using these new seeding methodologies.

$$PVE = \left(\frac{\text{Average Efficiency}}{\text{Average Best Fitness}} \right) \quad (5.4)$$

An interesting phenomena occurred during the testing of the Combo and HCI seeding methods. Figure 5.14 compares a single experimental run (that is indicative of the overall behavior identified). It is interesting to note that injecting optimized solutions into the initial population often resulted in the algorithm getting stuck in a local optima area and, as a result, the fmGA algorithm was essentially ineffectual and wasted precious computational time. Getting stuck in a locally optimized area is a very difficult problem to overcome mostly because the fmGA during its Juxtapositional phase is conducting local searches around the competitive template. As a result, the algorithm, more often than not, is unable to escape the local minima valley. On the other hand, both the α and β SSI seeding method, while not starting out with the best values, did allow the fmGA to be the dominating factor in finding better solutions (as depicted by the continual improvement in the best solution found), building block after building block. Note that the conformational dihedral angles for the various “good” minimum energy values found do not reflect similar positions in the PSP energy landscape and thus a local search technique should be employed with discretion.

5.8 Experiment 7: Incorporating the Epoch Idea

5.8.1 Results. Table 5.16 provides a comparison between the fmGA with and without the epoch modifications to the source code. Figure 5.15 provides a view of the fitness values per building block execution (recall from Chapter III that 1 epoch consists of the five building blocks from min_block_size to max_block_size (6 thru 10 and 16 thru 20 for [Met]-Enkephalin and Polyalanine respectively)).

5.8.2 Analysis. Clearly the fmGA with epochs found better solutions; however, it did so at a cost of nearly three times that of the algorithm without the epoch code turned on (see Table 5.16). The epoch results do improve upon the overall solution found; however, as we find better and better solutions, the epoch approach does not produce as well for the amount of time spent looking for a better solution. In fact, Figure 5.15 clearly shows that

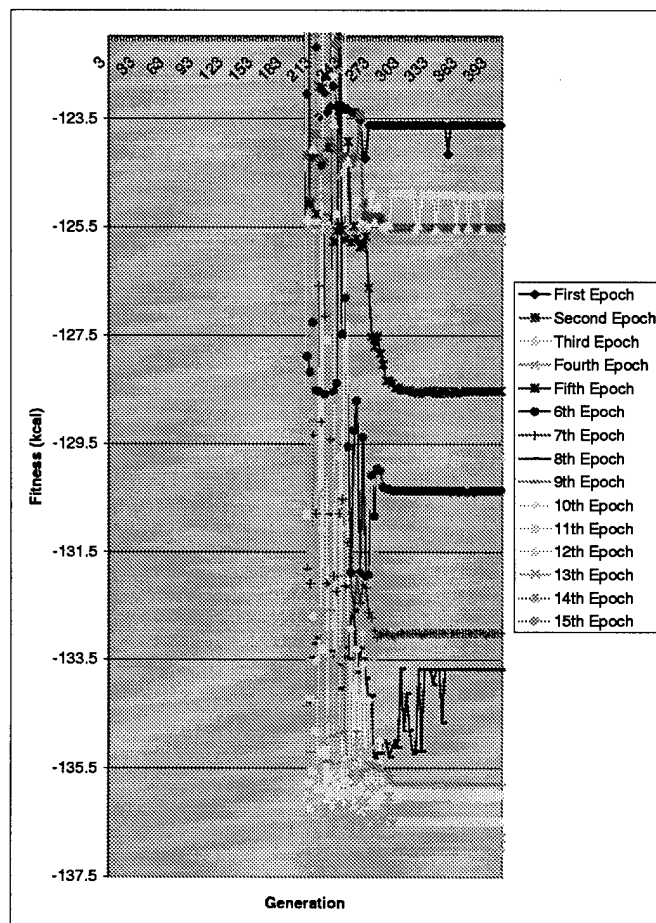


Figure 5.15. Fitness vs. Epoch

Table 5.16. Epoch Comparison

		1 epoch	1 epoch	3 epochs	3 epoch
		Fitness (kcal)	Time (seconds)	Fitness (kcal)	Time (seconds)
Polyallnine					
	Median	-119.519	1691.834	-127.766	4938.313
	Average	-120.884	1974.529	-128.778	4939.274
	Worst	-104.587	2278.932	-111.994	4968.354
	Best	-137.159	2584.226	-140.562	4911.262
	StdDev	9.867595	5.22049	8.548607	16.19961
[Met]-Enkephalin					
	Median	-22.8619	304.6031	-23.8956	895.3698
	Average	-22.4351	356.162	-23.7804	894.9455
	Worst	-16.8005	413.602	-20.938	901.3818
	Best	-25.1683	463.883	-26.3473	888.6199
	StdDev	2.157926	2.564546	1.52163	4.166535

as the building block size iterations are accomplished—(building block size iterations are repeated three times as the input parameter for epochs, which is an outside loop around the building block size iterative loop, is set to three) that the improvement in the solution from building block size to building block size decreases as time goes on. One other thing worth taking note of is the fact that after around generation 250 or so, the algorithm seems to not be able to find a better fitness for the remaining generations. A possible reason for this phenomena is that the fmGA during the Juxtapositional phase finds a local optima and from that point on is unable to escape. To avoid computational waste, perhaps a short-circuiting mechanism or re-hope mechanism should be employed to avoid this situation. In short, the incorporation of the epoch concept is beneficial in finding better fitness values; however, additional research is required to improve the performance hit one takes with implementing the epoch code.

5.9 Experiment 8: Short-circuiting the Energy Fitness Function

Before discussing the results of Experiment 8, it should be made clear that Equation (5.5) was used to derive the actual cost for a single evaluation on the chromosome defined for the Polyalanine structure.

$$T_{single_{eval}} = \frac{T_{Totaltime}}{Evals} \quad (5.5)$$

where

$T_{single_{eval}}$ is the elapsed time for a single evaluation (eval)

$T_{Totaltime}$ is the total time elapsed which is defined in Equation (5.6)

Evals is the number of CHARMM Energy fitness evaluations that were processed

$$T_{Totaltime} = \sum_{1..n} (T_{SingleEval_n} + C) \quad (5.6)$$

where C is defined as the remaining overhead associated with running the fmGA and for this experiment is assumed away as the same constant C is the same for both runs with

Processed	Type of Function	# of Evals	Time Elapsed (seconds)	Time/Eval (seconds)
GCT	Short-Circuited Fitness Function	39461	460.858	0.011679
Total Execution	Short-Circuited Fitness Function	28000	331.325	0.011833
GCT	Normal Fitness Function	41701	522.607	0.012532
Total Execution	Normal Fitness Function	28000	353.992	0.012643

Note: GCT = Generate Competitive Template

Figure 5.16. Effects of Short-circuiting the Results with Respect to Time

or without the short-circuiting function and when comparing them for their differences, the constant C is subtracted out.

Elapsed time for generating only the competitive templates which contained 28,000 evals was approximately 332.29 seconds while the total elapsed time for the entire algorithm was approximately 511.12 with total evaluations at 42,821.

Based on these numbers and Equation (5.5) we obtain a cost of anywhere from approximately 0.0119 to 0.01194 seconds per evaluation call.

5.10 Experiment 9: Effects of Ramachandran Constraints

5.10.1 Results.

5.10.2 Analysis. The tests for the Ramachandran constraints were not accomplished on the Polyalanine₁₄ protein as it was felt that a more generalized approach to incorporating this important domain knowledge into the algorithm was needed. The results for the Met-Enkephalin did prove to be worthwhile. There was a slight hit in the cost associated with implementing the code; however, it was less than a percent of the overall runtime of the fmGA.

Table 5.17. Results of Ramachandran Constraint Incorporation Into the fmGA

With Ramachandran Constraints	Met-Enkephalin Values (kcal/mol)	Polyalinine ₁₄ Values (kcal/mol)
Max	-22.721	—
Min	-28.075	—
Average	-26.167	—
Std Dev	1.606	—
Median	-26.114	—
Without Ramachandran Constraints		
Max	-21.261	—
Min	-26.089	—
Average	-23.822	—
Std Dev	1.463	—
Median	-23.392	—

Table 5.18. Optimal Solution Angular Values for [Met]-Enkephalin

Dihedral Angles (degrees)							
Residue	Φ	Ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr	-86	156	-177	-173	79	166	
Gly	-154	83	169				
Gly	84	-74	170				
Phe	-137	19	-174	59	-85		
Met	-164	160	-180	53	175	-180	-59

5.11 Final Observations

5.11.1 RMSD. The optimal dihedral values for the Met-Enkephalin protein and the *Polyalinine*₁₄ can be found in Tables 5.18 and 5.19 respectively.

Table 5.19. Optimal Solution Angle Values for *Polyalinine*₁₄

Dihedral Angles (degrees)				
Residue	Φ	Ψ	ω	χ
<i>Ala</i> ₁	65	(-30 thru -35)	180	60, -60, 120
<i>Ala</i> ₂	65	(-30 thru -35)	180	60, -60, 120
<i>Ala</i> ₃	65	(-30 thru -35)	180	60, -60, 120
\vdots	\vdots	\vdots	\vdots	\vdots
<i>Ala</i> ₁₄	65	(-30 thru -35)	180	60, -60, 120

Table 5.20. Best Dihedral Angular Values for [Met]-Enkephalin

Dihedral Angles (degrees)							
Residue	Φ	Ψ	ω	χ_1	χ_2	χ_3	χ_4
Tyr	97.734	-151.875	168.398	-170.156	-93.867	-39.727	
Gly	-141.328	111.445	-59.414				
Gly	47.109	-57.305	110.039				
Phe	37.266	-3.164	168.750	17.930	-166.289		
Met	98.438	-107.227	-90.352	67.500	-170.859	-107.227	70.313

Table 5.21. Best Dihedral Angular Values Found for Polyalanine₁₄

Dihedral Angles (degrees)				
Residue	Φ	Ψ	ω	χ
Ala ₁	-72.773	-0.352	106.875	79.805
Ala ₂	-56.25	88.945	80.508	-78.75
Ala ₃	-140.273	-39.727	138.516	-84.375
Ala ₄	-135	157.5	135.703	-105.82
Ala ₅	-84.727	-79.102	150.469	160.313
Ala ₆	-107.227	10.195	-162.422	-161.719
Ala ₇	179.648	145.195	157.1484	28.125
Ala ₈	179.648	5.273	-101.602	22.148
Ala ₉	-90.352	67.148	93.867	56.25
Ala ₁₀	11.25	90	177.890	-67.14
Ala ₁₁	179.648	-24.258	-88.954	22.5
Ala ₁₂	-90	22.5	159.609	78.75
Ala ₁₃	-87.539	-8.438	159.258	175.078
Ala ₁₄	-87.539	0	67.5	0

$$RMSD_{Worst} = \sqrt[2]{\sum_{i=0}^{NumberOfDihedralAngles} (OptimalAngle_i - FoundAngle_i)^2} \quad (5.7)$$

where Optimal Angle is given, Found Angle is derived, NumberOfDihedralAngles is determined by the protein under study and the particular representation used.

$$RMSD_{Best} = \sqrt[2]{\sum_{i=0}^{NumberOfDihedralAngles} (OptimalAngle_i - FoundAngle_i)^2} \quad (5.8)$$

where Optimal Angle is given, Found Angle is equal to Optimal Angle, NumberOfDihedralAngles is determined by the protein under study and the particular representation used.

The three methods in which we could use the RMSD equation are: 1) $\text{RMSD}_{\text{Backbone}}$ where the backbone dihedral angles are compared with either the mathematically derived angles or those that are found in nature and obtained via empirical study; 2) $\text{RMSD}_{\text{all-to-all-atoms}}$ where the coordinate of all atoms within the protein are compared to either the mathematically derived coordinates or those of the accepted known coordinates based on empirical study; or 3) RMSD_{C_α} where the coordinates of the atoms associated with C_α are compared to those coordinates obtained either mathematically or via empirical study.

Equations 5.7 and 5.8 represent the method in which to obtain the Worst and Best possible RMSD values for a given protein. For instance, suppose we decided to conduct a $\text{RMSD}_{\text{Backbone}}$ on a protein that contained 9 backbone dihedral angles (3 sets of ϕ , ψ , and ω dihedral angles) and its accepted values for each angle was 0 degrees. To determine the worst possible RMSD for that protein, we need to substitute the angle that would be exactly 180 degrees away from the Optimal dihedral angle. In this example we would use 180 degrees for each of the 9 dihedral angles found. Plugging in the numbers, we would obtain a 9.425 and 0.00 RMSD in radians for the worst and best RMSD, respectively—in fact the best RMSD is always 0.00.

For this thesis we chose to only conduct a $\text{RMSD}_{\text{Backbone}}$ comparison by using the dihedral angles from the best protein solution discovered and compare them with the empirically derived (or accepted) dihedral angles of the given protein. Based on these two equations (Equations 5.7 and 5.8) the best RMSD would be 0.00 and the worst $\text{RMSD}_{\text{Backbone}}$ is 12.167 and 19.619 for the Met-Enkephalin and Polyalanine₁₄, respectively.

Tables 5.20 and 5.21 represent the geometrical dihedral angles of the best solutions found during this thesis effort and is better than the results received for some of the work in this area. These angles result in a 5.26 and 7.59 $\text{RMSD}_{\text{Backbone}}$ (in radians) for each protein, respectively. This is comparable to conformation of the best Met-Enkephalin solutions found as Gates obtained a 5.47 $\text{RMSD}_{\text{Backbone}}$. A comparison between our best and Gates' best Met-Enkephalin solution [41] found results in a $\text{RMSD}_{\text{Backbone}}$ difference of 6.982 $\text{RMSD}_{\text{Backbone}}$ value which means our results were significantly different and the solution was in a very different area of the total solution space. The difficulty herein

is the fact that there are a very large number of local minima that can be found by a search algorithm and the difference between them and the accepted best is very small. No previous results have been published with respect to the Polyalanine₁₄ used in this study, thus no comparison is available.

Finally, it was mentioned that the $RMSD_{Worst}$ for both proteins was 12.167 and 19.619 with the larger value associated with the Polyalanine₁₄ protein. A direct comparison between these values and the values obtained, especially when we take into consideration Gates' results, we find that our work has improved results by almost 2% as Gates' best solution was normalized to .4496 and our results was normalized to .4323 (refer to Equation (5.9)).

$$RMSD_{Normalized} = \frac{RMSD_{Best-Solution-Found}}{RMSD_{Best} - RMSD_{Worst}} \quad (5.9)$$

where $RMSD_{Best-Solution-Found}$ is the calculated $RMSD_{Backbone}$ for Met-Enkephalin and $RMSD_{Best}$ and $RMSD_{Worst}$ were previously defined in the preceding paragraphs.

5.11.2 Kruskal Wallis Test. Kruskal Wallis test is a non-parametric test (distribution-free) used to compare three or more independent groups of sampled data. This non-parametric test makes no assumptions about the distribution of the data (e.g., normality). [102] This test is an alternative when the assumption of normality or equality of variance is not met. This, like many non-parametric tests, uses the ranks of the data rather than their raw values to calculate the statistic. Since this test does not make a distributional assumption. The test is set up as follows:

Test: The hypotheses for the comparison of two independent groups are:

H_o : The samples come from identical populations

H_a : The samples come from different populations

Notice that the hypothesis makes no assumptions about the distribution of the populations. These hypotheses are also sometimes written as testing the equality of the central tendency of the populations.

The test statistic for the Kruskal-Wallis test is H (see Equation (5.10)). This value is compared to a table of critical values for U based on the sample size of each group. If H exceeds the critical value for H at some significance level (usually 0.05), it means that there is evidence to reject the null hypothesis in favor of the alternative hypothesis. Note: When sample sizes are small in each group (< 5) and the number of groups is less than 4, a tabulated value for the Kruskal-Wallis test should be compared to the H statistic to determine the significance level. Otherwise, a Chi-square with $k-1$ (the number of groups-1) degrees of freedom can be used to approximate the significance level for the test. For all tests accomplished the sample size was 10 or more.

$$H = \frac{12}{n(n+1)} \left[\sum_{j=1}^k \frac{R_j^2}{n_j} \right] - 3(n+1) \quad (5.10)$$

where k = number of independent samples (or identical populations)

n_j = number of sample observations from the j th population

R_j = sum of ranks in the sample from the j th population (ranks are assigned by grouping all sample observations)

$n = \sum_{j=1}^k n_j$ = the total number of observations from the k samples

When comparing multiple data sets, we can use the results of the Kruskal Wallis test and apply the Tukey Multiple Comparison equation against the mean of the rankings assigned during the Kruskal Wallis test setup. Tukey's Multiple Comparison equation (see equation 5.11) provides a mechanism in which to determine if there is a critical difference between two sets of points via the ranking mechanism used in the Kruskal Wallis test.

$$CriticalDifference = Q_r \sqrt{\frac{MSW}{npergroup}} \quad (5.11)$$

where Q_r = value obtained from the Student T (two-tailed) table

MSW is the sum of the means squared times a weight factor

n per group = the number of samples per test set

1 - node pop-size = 200		Ranking	8 - node pop-size = 800		Ranking
	-27.67	1		-26.02	8
	-27.32	2		-25.79	9
	-27.31	3		-25.59	11
	-27.2	4		-24.64	12
	-26.79	5		-24.17	14
	-26.61	6		-23.64	15
	-26.43	7		-23.07	16
	-25.75	10		-22.71	18
	-24.41	13		-22.14	19
	-22.99	17		-20.43	20
Rank totals		68	Rank totals		142
Rank totals Squared		4624	Rank totals Squared		20164
Rank totals squared divided by number of samples		462.4	Rank totals squared divided by number of samples		2016.4
					2478.8
Kruskal Wallis' H value		7.822857	X table = .6635 for .99		Reject Ho Therefore Accept Ha

Figure 5.17. Kruskal Wallis Comparison Between Serial (Popsiz = 200) and 8-node (Popsiz = 800)

Using the above two statistical equations we now look at some of the experimental results on which to further draw conclusions about whether or not two samples are equivocal or not and thus, in the context of this research to ascertain whether or not the results of one algorithm dominate another.

5.11.2.1 Experiment 1: Serial vs Parallel. When we compared the serial run (population size of 200) with an 8-node run (population size 800), we obtained an H value slightly higher than the value obtained from the χ^2 distribution table for α set at .01. As a result, we reject the Hypothesis that these two sets of data have the same distribution and mean and accept the alternate hypothesis that these two sets are definitely different. With this in mind, we can infer that the serial run with a population set at 200 is a better run (with respect to minimizing the energy associated with the [Met]-Enkephalin protein) than the 8-node run.

When we compared the serial run (population size of 200) with a 32-node run (population size 4500), we obtained an H value significantly lower than the value obtained from the χ^2 distribution table for α set at .01. As a result, we accept the Hypothesis that these two sets of data have the same distribution and mean and reject the alternate hypothesis that these two sets are definitely different. With these results are unable to conclude that one run produced better results than the other.

1 node pop-size = 200		Ranking 32 - node pop-size = 4500 Ranking	
-27.67	3	-28.42	1
-27.32	4	-28.4	2
-27.31	5	-26.99	7
-27.2	6	-26.82	8
-26.79	9	-26.54	11
-26.61	10	-26.45	12
-26.43	13	-25.5	15
-25.75	14	-25.48	16
-24.41	19	-25.11	17
-22.99	20	-24.65	18
Rank totals	103	Rank totals	107
Rank totals Squared	10609	Rank totals Squared	11449
Rank totals squared divided by number of samples	1060.9	Rank totals squared divided by number of samples	1144.9
			2205.8
			0.02 < .6635
			Therefore
			Accept Ho
			0.022857 X table = .6635

Figure 5.18. Kruskal Wallis Comparison Between Serial (Popsiz = 200) and 32-node (Popsiz = 4500)

5.11.2.2 *Experiment 5: Searching for Secondary Structures.* To determine which of the tests had better results we attempt to analyze the results based on Kruskal Wallis testing. The results of the Kruskal Wallis analysis are shown in Figure 5.19. The figure attempts to compare the means of the rankings that were assigned to each sample point based on where they fit with respect to all other sample points. The figure is labeled to indicate the actual percentage the Secondary Search Parameter was set to and all positive numbers indicate that with that setting it had a lower mean (which indicates a mean rank position that was higher, by that value given in the table, than what it is compared to. While this compares each individual test set with each other, the Kruskal Wallis test demonstrates with a .999 confidence factor that the test sets are definitely different from one another (at least one test member must be different from one other test member in the set of tests). The H value as determined by the Kruskal Wallis test was 8.842.

Using the Table of differences of each test's ranking mean (Table 5.19) and Tukey's Multiple Comparison formula we can attempt to calculate with some degree of confidence which test proved to be dominant over the others. When we plug in the numbers required and set the α value to .01, we find that we need a Critical Difference in the means greater than 52.83. A quick look at the table shows us that comparatively speaking there are no mean differences greater than 19. In order to show some dominance, we have to increase

	0	0.9	0.8	0.7	0.6	0.5	0.4	0.2	0.1	Outranked
0	X	-13.3	1.7	0.8	-5.9	-4.1	-3	4.6	5.7	0
0.9	13.3	X	15	14.1	7.4	9.2	10.3	17.9	19	7
0.8	-1.7	-15	X	-0.9	-7.6	-5.8	-4.7	2.9	4	0
0.7	-0.8	-14.1	0.9	X	-6.7	-4.9	-3.8	3.8	4.9	0
0.6	5.9	-7.4	7.6	6.7	X	1.8	2.9	10.5	11.6	2
0.5	4.1	-9.2	5.8	4.9	-1.8	X	1.1	8.7	9.8	1
0.4	3	-10.3	4.7	3.8	-2.9	-1.1	X	7.6	8.7	0
0.2	-4.6	-17.9	-2.9	-3.8	-10.5	-8.7	-7.6	X	1.1	0
0.1	-5.7	-19	-4	-4.9	-11.6	-9.8	-8.7	-1.1	X	0
Outranked	1	0	1	1	0	1	1	2	3	X

Figure 5.19. Comparison Between Ranking Means of Secondary Structure Search Results

the α value to .4, which lowers the Critical Difference to 13.99. With a Critical Difference of 13.99, we see that only test “.9” scored above the Critical Difference factor of 13.99 and it did so 4 out of 8 times. With this in mind, it is probably acceptable to deduce that the parameter value of “.9” will provide the overall best results 60% of the time. Further conclusions cannot be drawn based on these results.

This set of tests was accomplished to provide the reader an alternative method in which to view the data contained herein. The work accomplished here entails a great number of factors that cannot be coalesced into a single data point and thus it was not deemed overly important to run each of the test results against the Kruskal Wallis test.

5.12 Summary

This chapter focused on the test results and what they actually meant. Each test was designed to gain insight into whether or not the code modifications were beneficial or not. As previously identified, we deemed a change in the code beneficial if there was an improvement in the overall efficiency of the algorithm with respect to performance and the effectiveness in finding a “better” solution. In most of the tests, there were clear cut improvements in either performance and/or effectiveness; however, with that the most important results of this work is the fact that we have improved the algorithm to locate a better conformation of the Met-Enkephalin (see previous section’s discussion of RMSD). Regardless of the results, there is much work to be done and the following chapter will provide a list of possible directions in which this research can go. Finally, it is hoped that through the use of the $RMSD_{Backbone}$, Kruskal Wallis testing, Tukey’s Multiple

Comparison, numerous tables and figures, and new metrics to measure performance, we have provided a glimpse of the overall Protein Structure Prediction problem and our efforts and contributions in moving research in this area forward.

VI. Conclusions

6.1 Observations

It is our belief that the results of this thesis have moved our search, in finding a sound algorithmic approach to finding the “optimal” conformation of a given protein, a little further along the road. The fact that we are having difficulties in finding the optimal solution for small proteins of less than 150 atoms means we have a long way to go, especially since we know many proteins contain hundreds of thousands of atoms.

During our study we devised a number of new metrics (Performance vs. Effectiveness (PVE), see Equation (5.4)). With which to compare/constrast various test results. We provide an alternative to Deerman’s visualization [25] of the landscape (see Section 2.2.7). We incorporated secondary structure information into various portions of the fmGA with promising results. We improved the processor utilization of the fmGA in a heterogeneous environment enabling the algorithm to conduct additional localized searches which; in turn, resulted in a significant improvement in the fitness of the best solution found.

The overall goal of this thesis effort was to improve the fmGA by incorporating secondary structure information and/or through other code modifications. All code modifications were met with varying success with respect to the the code without the enhancements with the only exception being the Short-Circuiting of the Fitness Function code modification. Our best solution for the Met-Enkephalin protein was shown to have nearly a 2% better $\text{RMSD}_{\text{Backbone}}$ value than what Gates reported in his thesis [41]. Although our set of dihedral angles were significantly different than the scientifically accepted values for the Met-Enkephalin protein there is supporting evidence that there are a number of equally low minimized values for different geometrical conformations for a given protein and some scientists argue further that the conformation state might be a metastable state with high barriers, or it might just be the lowest local minimum that is kinetically accessible from most of the protein’s energy space [107]. In other words, our solution may have a better fitness, but is impossible to fold to in nature.

6.2 Future Considerations

Although a number of ideas and enhancements have been integrated into the fmGA, there is more that can be done. The work of this research has provided a better algorithm; however, we still desire a highly reliable methodology of searching for the conformation of a protein (large or small). The main reason for the problem is the fact that there are many local optima, as we have already discussed, and the fmGA continues to have a difficult time in escaping them.

The following itemized list are some of the things that can be tried in hopes of finding a very effective means of identifying the conformation of a given protein:

1. Incorporate some sort of computational steering method (see [109]) into the fmGA. With the biologist's ability to recognize patterns this sort of ongoing interaction between the computer algorithm and the human can only help the algorithm get "unstuck" from a local optimal.
2. Change the fmGA such that it runs until no better solution is found after a given generation and after a given epoch. The current implementation stops at generation 400 (or whatever the user specifies) and after 3 epochs. This idea accomplishes two things; first, it eliminates computational waste such as was shown and discussed in Section 5.8. Secondly, it should provide for better results as the algorithm would then only stop once no better solution has been found for a given length of time, instead of stopping regardless of whether or not the algorithm is on the road to better results.
3. Take advantage of the lessons learned by Kaiser [69] and Incorporate Real Values into the fmGA. This would remove the effect of disrupting good building blocks within a dihedral angle. The current implementation allows the angle building blocks to become disrupted (recall our chromosome has 10 bits per angle, but a cut-and-splice operation can occur anywhere within the chromosome, resulting in the breaking apart of a "good" dihedral angle).
4. Analyze the 9 sub-components of AFIT's CHARMm Energy Fitness function with the intent of grouping them into 2 or more sets. Once this has been done, then the fmGA should be modified such that it is now multi-objective in that it compares

each of the sets of energy functions (each containing different physical aspects of the interactions of all the atoms) to each other. Additionally, the incorporation of multi-objective fitness functions could allow the fmGA to be expanded to take into account other objectives such as identifying if a secondary structure exists or not.

5. The incorporation of uniquely derived competitive templates and allowing the fmGA to work with more than one competitive template. The idea would be to generate competitive templates in more than one fashion (based on optimal secondary structure dihedral angular values, or generated via a panmictic approach, etc.)
6. Determination of optimal parameter settings to obtain the best results. Recall in the discussion that the current settings for many of the parameters are based on purely empirical results. This means that there is a high chance that the current parametric settings are not optimal or even close and they most probably differ for each protein under study.
7. Modify the AFIT CHARMM Fitness function to allow a given chromosome to be evaluated only on the backbone by holding the ω and χ dihedral angles to some constant. It is our belief that the reason the fmGA cannot get out of a local optimal is because the atoms associated with the side chains come in too close contact with other atoms causing a drastic increase in the fitness value and thus not allowing the algorithm to move its search to a better location.
8. Modify the fmGA code such so that it is able to handle Ramachandran constraints "generically" for any given protein based solely on the number of residues it contains. The current implementation only contains Ramachandran constraints for the [Met]-Enkephalin as there was no time to calculate the constraints for the Polyalanine₁₄ protein. Since the chromosome structure that the AFIT CHARMM model uses defines n-1 residues in the first (n*30) bits the idea would be to generically handle the Ramachandran constraints only for the first n-1 set of dihedral angles associated with the Total Number of Residues the protein contains minus 1.
9. Incorporate a "pre-parsing" algorithm that helps identify possible secondary structures and their locations with respect to the protein chain (homology) [18, 68, 103,

14, 125, 98, 138, 128] (see section 2.2.1). Recall that current techniques used to identify secondary structure a priori have resulted in a success rate of nearly 75%. Pre-parsing could be used as a mechanism in which to develop the competitive template or seed the population with chromosomes that have secondary structure angles associated with the results of the pre-parsing algorithm.

10. Further research in the area of short-circuiting the energy fitness function. Specifically the ability to initially conduct a course-grain search by only using a few of the subfunctions that make up the AFIT CHARMM Energy fitness function, and as the search continues, additional subfunctions are added to the evaluation process. It was obvious from our results that current method of short-circuiting the energy fitness function after the very first subfunction completes if the fitness is beyond a threshold value that it forces exploration in parts of the landscape which do not hold very good local optimal points. This effect needs to be better understood before further action in this area can be taken.

It is our hope that this document has provided the reader insight into the Protein Structure Prediction problem and the fast messy Genetic Algorithm and that the reader is further encouraged to take up the Grand Challenge described herein.

Appendix A. Porting from NX to Message Passing Interface (MPI)

A.1 Introduction

Prior to this research effort, AFIT's pfmGA PSP software was written to run on the Intel Paragon located at the Material Resource Center (MRC) using NX communication hooks and also for the SP2 computer using MPI communication hooks by Gates and Merkle [93]. Unfortunately only the NX version of the software was located for this research effort and the fact that the Intel Paragon supercomputer is no longer available led us to have to re-parallelized the software using MPI communication constructs. This was the first step taken for this thesis effort. In a nutshell, all the synchronous communications that was coded using the NX communications package had to be converted to MPI communication calls. The code was then executed and tested on all four platforms identified in Appendix B.

A.2 NX communications system calls

The system calls and library routines (referred to collectively as "system calls") that let you access special capabilities of the Intel supercomputer. The following list are only some of the communications functions that are provided:

1. Create and control parallel applications and partitions.
2. Exchange messages between processes.
3. Get information about the computing environment (such as, the number of nodes in the current application)

Table A.2 list the original NX communication function calls used to implement parallel code on the Intel Paragon.

A.3 MPI communication routines

MPI is a message-passing library specification that contains a message-passing model, is not a compiler specification, and is not a specific product. It is useful for parallel computers, clusters, and heterogeneous networks. It contains a full range of features and is

Table A.1. NX Function Calls [65]

- CRECVX()	- Posts a receive for a message and blocks the calling process until the receive completes (Synchronous receive). Allows user to specify either node and process type to receive from or receive from any node or process type. (Synchronous receive).
- CRECV()	- Posts a receive for a message and blocks the calling process until the receive completes (Synchronous receive).
- CSEND()	- Sends a message and blocks the calling process until the send completes (Synchronous send). Allows user to specify both node and process type in which to send to or the option to send to all nodes and all process.
- DCLOCK()	- Gets elapsed time in seconds since the node was booted-has an accuracy of 100 nanoseconds.
- GCOL()	- Collects contributions from all nodes and concatenates them in node order. Results are collected and returned to all nodes.
- GCOLX()	- Collects contributions of known length from all nodes and concatenates them in node number order. Results returned to all nodes.
- GDLOW()	- Determines the minimum "double precision" value across all nodes and returns that value to all nodes.
- GILOW()	- Determines the minimum "integer" value across all nodes and returns that value to all nodes.
- GSYNC()	- Synchronize all node processes in an application.
- IRECV()	- Posts a receive for a message and returns immediately (Asynchronous receive). Receives from any node or process type.
- ISEND()	- Sends a message and returns immediately (Asynchronous send). User must specify to send to one specific node or process type or to all nodes and all process types (or any combination of the two).
- NUMNODES()	- Gets the number of nodes in an application.

designed to permit the development of parallel software libraries, as well as, provide access to advance parallel hardware for end users, library writers, and tool developers.

Message passing is a well understood process that can be efficiently mapped to hardware and supports a multitude of applications. It is very portable, though vendor specific implementations may not be. Few systems offer the full range of desired features such as modularity (for libraries), access to peak performance, portability, heterogeneity, sub-groups, topologies, and performance measurement tools.

Some of the general features are [113]:

Communications combine context and group message security
Thread safety
Point-to-point communications
Collective operations—built-in and user-defined
Application-oriented process topologies
Profiling tools

MPI has over 125 functions available though one can get by with only six of those and successfully implement a parallel program. MPI Communication calls described in Table A.3 were used to port the pfmGA PSP software from the INTEL NX communications to the MPI communications standard.

While there are other communications software packages that could have been used to allow the software to run in parallel, MPI was chosen for the following reasons:

4) Specific NX to MPI code modification made to the pfmGA PSP software.

a) Initialization of the MPI Communications Protocol

In order to utilize MPI, one must first initialize it. To initialize MPI the MPI.Init() function is called. It was passed two parameters "&argc, &argv". These two arguments allow the user to pass in information from the command line. In almost all implementations that were looked at during this conversion process, both MPI.Comm_size and MPI.Comm_rank were called to store information about the number of nodes in the parallel run and what rank a respective parallel implementation of the code is. Because this information is extremely important to conduct certain processes for specific nodes and/or

Table A.2. MPI Functions [113]

- MPI.Allgather()	- All nodes collect contributions from all other nodes and concatenates them in node number order.
- MPI.Allreduce()	- All nodes collect the specified data from all other nodes, determines the which value meets the defined requirement, i.e. max/min, and places the results into each node's respective buffer.
- MPI.Barrier()	- Synchronizes all nodes to a point in the program
- MPI.Init()	- Required at the beginning of the program where parallel operations will take place.
- MPI.Comm_size()	- Used to determine the number of processors that have been allocated and are being used to implement the program in parallel.
- MPI.Comm_rank()	- Provides the processor that calls this function its relative rank with respect to all other nodes running the same parallel code.
- MPI.Wtime()	- Provides a incremental time based on the start of the parallel code.
- MPI.Recv()	- Provides a blocking receive method that can receive from any or a specified node or group.
- MPI.Send()	- Provides a blocking sending method that can send to all or single node or designated group.

an attempt to send or receive something iteratively this was incorporated into the pfmGA PSP software.

b) Initial Loading of the Parameter File

The first thing the software did upon startup was to read in information from the parameter files, as well as, all molecular specific source data files. The parameter file, once read in and stored appropriately, was passed via NX csend() communication call to all other nodes running in parallel. This code was removed and instead of replacing it

Table A.3. Reasons for Choosing MPI as the Parallel Communications Medium

1. AFIT has nearly 4 years of programming with MPI as the underlying communications package
2. MPI is readily available on all three of AFIT's parallel computer systems
3. MPI is support across a multitude of platforms
4. The Protein Structure Prediction data representation does not end itself to a parallel model
5. MPI is readily supported for the C/C++ software language
6. MPI has equivocal communication routines to of NX

with an appropriate MPI communications call, it was decided that merely reading in the information after the MPI communications was initialized would be sufficient. Since the Sun Workstations, Pile of PCs, the SP2, and SP3 all have a transparent directory structure (i.e. regardless of the processor running the software, all parallel implementations of the software see the same directory information that the program started in. As a result, once MPI Init() has been called, any file opening and subsequent reading is done by all processors and since we do not modify the file, the file I/O is pretty quick and practically transparent to the overall operation of the pfmGA PSP software.

c) Generating the Competitive Template

Once the loading of parameters and molecular data, as well as, the creation of appropriate memory variables, the next step in the algorithm is to generate a competitive template. Each node in the parallel implementation generates a competitive template. Once generated, all nodes will call the MPI_Allreduce() function passing their respective competitive template's fitness evaluation value. The MPI_Allreduce() function will return to all nodes, the fitness value that is the lowest of all. Later, that node that had the lowest fitness evaluation value for their respective competitive template will initiate a MPI_Send() function loop and send the actual string configuration of the competitive template to all other nodes, iteratively-this is where the results from both the MPI_Comm_size function and MPI_Comm_rank come into play.

d) Transferring Population members

Following the Create_Initial_Population() and Conduct_Primordial_Phase() function calls, the algorithm does a series of Collect_Population(), Exchange_Population, Immigrate(), and Emmigrate() function calls. Each of these functions require MPI communications of one of the following:

MPI_Recv MPI_Send MPI_Allgather MPI_Allreduce

- Each of these function calls pass appropriate information to other node(s) required by the pfmGA algorithm. For additional information about the exact MPI communication calls made, please refer to the source code.

Appendix B. Parallel Platforms and Computing

B.1 Introduction

AFIT currently has two distinct parallel computing platforms; the cluster of Workstations and the Pile of PCs. In addition, Wright Patterson Air Force Base's Material's Lab has the MSRC, which has three distinct supercomputers (SP2, SP3, and SG2000). Only the Cluster of Workstations and Pile of PC's, and SP2 and SP3 were used to conduct both serial and parallel fmGA PSP software runs.

1) AFIT Parallel Platforms

The three tables (B.1, B.2, and B.3) identify individual machine by performance and machine name for AFIT's two parallel environments. There are a few systems that have more than a single processor and are identified appropriately.

The communications backbones for both AFIT parallel platforms are quite unique with respect to each other. The Hawkeye platform generally has a 100MBps Ethernet capacity; however, a 10MBps Ethernet limits the connection between the Hawkeye lab and the parallel lab and the internal communications capacity within the parallel lab is also 10MBps. On the other hand, the Pile of PCs, has a 1GBps Ethernet backbone that connects two 100MBps Ethernet sub-nets (which currently constitutes the bottleneck for the Pile of PCs). 100MBps and 10MBps translate to 12.5 and 1.25 MBytes per second, respectively.

One single factor has had a major part in the conducting of experiments--available disk space. Unfortunately, the Hawkeye platform has only 600 to 800 MBytes of space available. This is not nearly enough, as the number of students utilizing the Hawkeye system is quite large and their respective data sets can and do tax the system and has resulted in complete failure. Like the Hawkeye platform, the Pile of PCs had disk space limitations also, though work to increase the available disk space to application users was increased from 2 GBytes to 6 GBytes.

2) MSRC Parallel Platforms

Table B.1. Cluster of Workstations (Sun Microsystems)

Name	Model	processor	# of CPUs	Physical RAM (MB)	NIC speed (Mb/s)	OS	ip addr
allele	Ultra1 170	Ultra Sparc 170	1	128	10	2.7	129.92.9.177
amino	Ultra 1 200	Ultra Sparc 200	1	128	100 capable	2.7	129.92.9.179
atilla	Ultra 1 170	Ultra Sparc 170	1	256	10	2.7	129.92.5.35
bach	Ultra 1 170	Ultra Sparc 170	1	320	10	2.7	129.92.5.34
barruc	Ultra 1 170	Ultra Sparc 170	1	128	100 capable	2.6	129.92.5.44
belle	Sparc 5 model 85	Microsparc 2 85	1	96	10	2.6	129.92.5.182
bucephalus	E450 Model 4400	Ultra Sparc 2 400	4	2048	100	2.7	129.92.5.6
chip	Ultra 10 Model 300	Ultra Sparc 300	1	128	100	2.7	129.92.5.45
clovia	Sparc 5 model 110	Microsparc 2 110	1	96	10	2.6	129.92.5.185
codon	Ultra 1 170	Ultra Sparc 170	1	128	10	2.7	129.92.9.176
cordelia	Ultra 5/10 Model 440	Ultra Sparc 2i 440	1	1024	100	2.7	129.92.5.136
darwin	Ultra 1 170	Ultra Sparc 170	1	128	100 capable	2.7	129.92.9.178
delenn	Ultra 2 Model 2170	Ultra Sparc 168	2	128	100 capable	2.7	129.92.5.33
doc	Sparc 5 model 110	Microsparc 2 110	1	48	10	2.6	129.92.5.188
draal	Ultra 2 Model 2170	Ultra Sparc 168	2	128	100 capable	2.7	129.92.5.43
ernie	Ultra 2 Model 1200	Ultra Sparc 200	1	128	100 capable	2.7	129.92.5.177
euclid	Sparc 5 model 85	Microsparc 2 85	1	32	10	2.6	129.92.5.181
fuggles	Ultra 1 170	Ultra Sparc 170	1	256	100 capable	2.6	129.92.5.184
gene	Ultra 1 170	Ultra Sparc 170	1	128	10	2.7	129.92.9.175
grimm	Ultra 1 170	Ultra Sparc 170	1	192	100 capable	2.6	129.92.5.42
hack	Sparc 5 model 110	Microsparc 2 110	1	48	10	2.6	129.92.5.187
hawkeye	Ultra 2 Model 2200	Ultra Sparc 200	2	128	100 capable	2.7	129.92.5.4
hilbert	Ultra 1 170	Ultra Sparc 170	1	128	100 capable	2.7	129.92.5.37
hobbes	Sparc 20 model 50	Sparc 50	1	64	10	2.6	129.92.5.186
huygens	Ultra 1 170	Ultra Sparc 170	1	128	10	2.7	129.92.5.40
ivanova	Ultra 1 170	Ultra Sparc 170	1	128	10	2.7	129.92.5.38
joel							129.92.5.48
katharina	Ultra 5/10 Model 440	Ultra Sparc 2i 440	1	1024	100	2.7	129.92.5.140
kramer	Sparc 20 model 60	Sparc 60	1	224	10	2.6	129.92.5.183
lacertae	Ultra 1 170	Ultra Sparc 170	1	128	10	2.7	129.92.5.39
leonardo							129.92.22.3
mach	Sparc 20 model 70	Sparc 70	1	96	10	2.6	129.92.5.191
maddie	Ultra 1 170	Ultra Sparc 170	1	192	10	2.7	129.92.5.41
maria	Sparc 20 model 50	Sparc 50	1	64	10	2.6	129.92.5.96
maxwell							129.92.5.133

Table B.2. Cluster of Workstations (Sun Microsystems) Cont.

Name	Model	processor	# of CPUs	Physical RAM (MB)	NIC speed (Mb/s)	OS	ip addr
mccoy							129.92.8.1
merlin	Sparc 20 model 50	Sparc 50	1	128	10	2.7	129.92.9.186
michelangelo							129.92.22.12
miranda	Ultra 5/10 Model 440	Ultra Sparc 2i 440	1	1024	100	2.7	129.92.5.138
monet							129.92.22.6
montross	Ultra 1 170	Ultra Sparc 170	1	256	100 capable	2.6	129.92.5.180
newton	Ultra 60 Model 2300	Sparc 2 300	2	512	100 capable	2.7	129.92.5.129
nimrod							129.92.20.8
nina							129.92.5.50
nova							129.92.5.134
nyquist	Ultra 1 170	Ultra Sparc 170	1	192	10	2.7	129.92.5.36
ophelia							129.92.5.137
phoenix							129.92.11.1
pinna							129.92.5.8
poisson							129.92.5.190
portia	Ultra 5/10 Model 440	Ultra Sparc 2i 440	1	1024	100	2.7	129.92.5.139
prowler	Sparc 5 model 85	Microsparc 2 85	1	48	10	2.7	129.92.5.179
quasar	Sparc 20 model 70	Sparc 70	1	128	10	2.7	129.92.5.131
quvat	Sparc 20 model 50	Sparc 50	1	128	10	2.7	129.92.5.132
raphael							129.92.22.38
rembrandt							129.92.22.2
rufus							129.92.5.49
sarge							129.92.5.46
seurat							129.92.22.11
shaman	Ultra 1 170	Ultra Sparc 170	1	256	10	2.7	129.92.5.178
shogun	Sparc 10 Model 40	Sparc 40	1	192	10	2.6	129.92.5.192
sipl1							129.92.5.1
sipl2							129.92.5.2
sipl3							129.92.5.3
sophie	SparcServer 1000	Sparc 50	2	192	100 capable	2.7	129.92.5.5
spock							129.92.22.9
tong	Sparc 10 Model 40	Sparc 40	1	64	10	2.6	129.92.5.95
trapper	Ultra 2 Model 2200	Ultra Sparc 200	2	320	100 capable	2.7	129.92.5.7
vangogh							129.92.22.32
viper	Ultra 1 170	Ultra Sparc 170	1	128	100 capable	2.7	129.92.5.130
walt	Sparc 5 model 85	Microsparc 2 85	1	80	10	2.7	129.92.5.47
whitestar	Sparc 5 model 110	Microsparc 2 110	1	96	10	2.6	129.92.5.189

Table B.3. File of PCs

Name	Processor	Physical RAM (MB)	Network speed (Mb/s)	OS#1	OS #2	IP ADDR	NP
Win2000	dual Pentium III/Xeon 550	784	1000		W2k	172.8.0.1	2
Linstar	dual Pentium III 600	512	1000	RH6.2		172.8.0.2	2
abc-a3	Pentium III 600	384	1000	RH6.1	W2k	172.8.0.3	1
abc-a4	Pentium III 600	384	1000	RH6.1	W2k	172.8.0.4	1
abc-a5	Pentium III 600	384	1000	RH6.2	W2k	172.8.0.5	1
abc-b5	Pentium III 600	384	100	RH6.1	W2k	172.8.1.5	1
abc-b6	Pentium III 600	384	100	RH6.1	W2k	172.8.1.6	1
abc-b7	Pentium III 600	384	100	RH6.1	W2k	172.8.1.7	1
abc-b8	Pentium III 600	384	100	RH6.1	W2k	172.8.1.8	1
abc-b9	Pentium III 600	384	100	RH6.1	W2k	172.8.1.9	1
abc-b10	Pentium II 400	256	100	RH6.1		172.8.1.10	1
abc-b11	Pentium II 400	256	100	RH6.1		172.8.1.11	1
abc-b12	Pentium II 400	256	100	RH6.1		172.8.1.12	1
abc-b13	Pentium II 333	256	100	RH6.1		172.8.1.13	1
abc-b14	Pentium II 400	128	100	RH6.1		172.8.1.14	1
abc-b15	Pentium II 450	256	100	RH6.1		172.8.1.15	1
abc-b16	Pentium II 400	128	100	RH6.1		172.8.1.16	1
abc-b17	Pentium II 400	128	100	RH6.1		172.8.1.17	1
abc-b18	Pentium II 333	128	100	RH6.2		172.8.1.18	1
abc-b19	Pentium II 333	128	100	RH6.2		172.8.1.19	1
abc-b20	Pentium II 333	256	100	RH6.2		172.8.1.20	1
abc-b21	Pentium III 1GHZ	256	100	RH6.2	W2k	172.8.0.21	1
abc-b22	Pentium III 1GHZ	256	100	RH6.2	W2k	172.8.0.22	1
abc-b23	Pentium III 1GHZ	256	100	RH6.2		172.8.0.23	1
abc-b24	Pentium III 1GHZ	256	100	RH6.2		172.8.0.24	1
abc-b25	Pentium III 1GHZ	256	100	RH6.2		172.8.0.25	1
abc-b26	Pentium III 933 HZ	256	100	RH6.1		172.8.0.26	1
abc-b27	Pentium III 933 HZ	256	100	RH6.1		172.8.0.27	1
abc-b28	dual Pentium III/xeon 550	512	100	RH6.1		172.8.0.28	2
abc-b29	dual Pentium III/xeon 550	256	100	RH6.1		172.8.0.29	2
micron	Pentium 200	64	100	RH6.2		172.8.1.30	
abc-32	Pentium II 266	128	100	RH6.1		172.8.1.32	
HP4si						172.8.0.31	

As indicated before the SP2 and SP3 parallel computer systems were used to test the pfmGA PSP software. The following provide specific hardware details of each system.

SP3 Data:

1. 375MHz SMP ThinNode 4-Way processor
2. Total FLOPS = 1500Mflops
3. L1 Cache= 64Kbytes
4. L2 Cache=8Mbytes
5. Bandwidth= 130MB/sec
6. TCPIP BW=123MB/sec for best SMP nodes
7. 4 Processors (375MHz each) per node
8. 132 SMP nodes
9. 4 Gbytes RAM
10. 4 Gbytes Swap per node
11. Only 130 nodes available for batch processing

SP2 Data:

1. 133MHz SMP ThinNode processor
2. Total FLOPS = ???Mflops
3. L1 Cache = 64Kbytes
4. L2 Cache = 8Mbytes
5. Bandwidth = 130MB/sec
6. TCPIP BW=123MB/sec for best SMP nodes
7. 4 Processors (375MHz each) per node
8. 132 SMP nodes
9. 4 Gbytes RAM

10. 4 Gbytes Swap per node

11. Only 130 nodes available for batch processing Figure

B.2 Parallel Computing

This appendix contains background material on parallel computing, most of which has been presented in previous AFIT theses, Merkle [88]. Two distinct but interdependent aspects of parallel computing are presented. Section B.2.1 considers issues related to the design and implementation of parallel computer architectures. Section B.2.2 examines the design and implementation of algorithms which exploit application parallelism.

B.2.1 Parallel Architectures. The vast majority of computer architectures in common use are based on the organization proposed by Von Neumann in the 1940s, in which a single memory area is used to store both instructions and data. Such architectures are referred to as Von Neumann-based. Von Neumann-based parallel processing systems can be categorized as Multiple Instruction Multiple Data (MIMD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), or Single Instruction Single Data (SISD). A special case of the MIMD category is the Single Program Multiple Data (SPMD) paradigm. Other architectures exist, but their use is primarily limited to research. The majority of commercially available parallel architectures are either SIMD or MIMD.

During any given instruction cycle, all of the processors of a SIMD architecture execute the same instruction, using different data. In order for the instructions to be applicable to the data on all the processors, they must be more general and therefore less powerful. As a result, the individual processors have small instruction sets, making them relatively inexpensive, so that it is cost effective for SIMD architectures to include large numbers of processors. SIMD architectures with 64,000 processors are fairly common. The tradeoff is that for a fixed amount of memory, there is less memory per processor.

In contrast, the processors of a MIMD architecture act independently, and can take advantage of more powerful instructions. Each processor is more expensive, so that MIMD architectures are typically implemented with fewer processors than SIMD architectures. This means that each processor can be allocated more memory.

A single processor and its allocated memory are together called a *node*. The relative computational power of each node in a parallel architecture is often referred to as the *granularity* of the architecture. Most SIMD architectures are categorized as *fine grained* because they have a large number of nodes, each of which has a simple processor with a small amount of memory. In contrast, most MIMD architectures are categorized as *coarse grained* because they have a relatively small number of nodes, each with a powerful processor and significant memory.

Some architectures allow processors to access memory allocated to other processors, or simply allow all the processors to access a single global memory. Such architectures are referred to as *shared memory* architectures. Processors within such architectures can communicate data by storing it in memory which is accessible to the receiving processor. Most SIMD architectures are in this category. Most MIMD architectures, on the other hand, are *distributed memory*, meaning that processors cannot access memory allocated to other processors. These architectures are also referred to as *message passing*, because the processors communicate via communication links. This type of communication is generally very slow relative to other processor activities.

Parallel architectures can also be categorized according to their *interconnection topology*, or *network*, which defines the other processors to which each processor can communicate data. A common interconnection topology for SIMD architectures is a *2-D mesh*, in which the processors are arranged, logically if not physically, in a two dimensional array. A 4×4 mesh is shown in Figure B.1. Mesh interconnection networks allow each processor to communicate data to each of the four processors at its sides. Well known examples of such systems are the Connection Machine, which is manufactured by Thinking Machines, Inc., and the Paragon, which is manufactured by Intel.

A common interconnection topology for MIMD architectures is a *hypercube*. Hypercube architectures have a *dimension*, N , and have 2^N processors. A hypercube of dimension 4 is shown in Figure B.2. Each processor is directly connected to, and can communicate data to N other processors in a single step. Any processor can communicate data to any other processor in no more than N steps. One of many examples of a com-

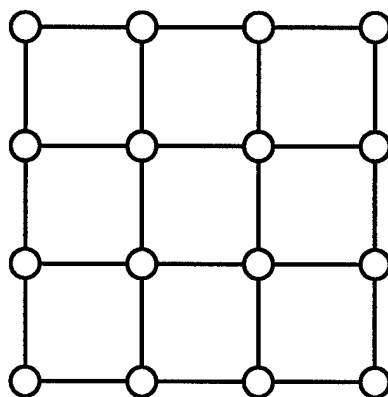


Figure B.1. 4×4 Mesh Interconnection Network

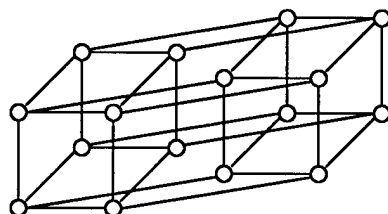


Figure B.2. Dimension 4 Hypercube Interconnection Network

mercially available hypercube architecture is Intel Corporation's parallel supercomputer, the iPSC/i860.

B.2.2 Parallel Algorithms. Software development for parallel architectures is fundamentally different than for sequential architectures [12]. The primary question in developing parallel software is whether to design parallel algorithms and implement them directly, or to implement sequential algorithms and then parallelize them. For most cases in which there is no sequential software predecessor, and even in some cases where there is such a predecessor, the first approach likely results in better performance.

Several algorithm properties can lead to performance improvements when the algorithms are implemented in parallel. The two most common such properties are data parallelism and control parallelism[84]. The former describes a situation in which an algorithm processes multiple data items in the same way, and the actions taken for any particular data item do not depend on the results of processing other data items. The latter is present when two distinct operations on the same data do not depend on each other. Another form

of parallelism, sometimes referred to as “trivial” parallelism, is present when two separate activities, which share neither control nor data, can be executed simultaneously.

Chandy and Misra propose an architecture independent method for the description of an algorithm[12]. A UNITY (Unbounded Nondeterministic Iterative Transformations) program describes the requirements for a process. It does not specify the order of operations or the mapping of operations to processors. Thus, a UNITY program may be mapped to any architecture, whether it be sequential, asynchronous shared-memory, or distributed memory. The description of a mapping describes how the UNITY program is executed on the target architecture. Mappings for particular classes of architectures exhibit common characteristics. The target architecture in this study is a hypercube, which is a distributed memory (DM) system. Chandy and Misra describe DM systems formally as consisting of a fixed set of processors, a fixed set of communication channels, and a memory for each processor[12:83]. As such, a mapping to such an architecture must

- allocate each statement in the program to a processor;
- allocate each variable to either a memory or a channel;
- specify the control flow for each processor;
- allocate at most one variable, which is of type sequence, to each communication channel;
- be such that a variable which is allocated to a channel is referenced in statements which are allocated to exactly two processors.

Furthermore, the statements allocated to one of the processors which reference a channel variable may only modify the variable by appending an item to the sequence. They may only do so when the sequence is of length less than a constant buffer size. Finally, the statements allocated to the other processor which references the channel variable may only modify the variable by removing the first item in the sequence. They may only do so when the length of the sequence is greater than zero. ’

Appendix C. IBM's "Blue Gene" Supercomputer

C.1 Computer

In December of 1999, IBM startled the parallel computing industry by announcing their \$100 million research initiative to build the worlds fastest parallel processing machine. The new computer, nicknamed "Blue Gene" will be capable of more than one quadrillion operations per second (one petaflop)—approximately 2 million times more powerful than today's top desktop PC.

Blue Gene will initially be used to model the folding of human proteins (Molecular Dynamics) making this IBM's first computing "grand challenge" since the Deep Blue experiment. According to Moore's Law achieving petaflop-scale performance is still 15 years away; however, IBM is determined to cut this time down to five years and they believe their radically new approach can do it. IBM researchers are calling their experimental design "SMASH," which stands for Simple, Many, and Self-Healing. "Simple" refers to the elemental architecture; "Many" describes Blue Gene's one million processors working in parallel; and "Self-Healing" means greater fault-tolerance and system stability. This experimental architecture should allow more processors to be placed in less space with reduced power requirements, resulting in more operations overall. IBM researchers expect their design to be able to achieve around 8 million simultaneous threads of communications—a tremendous difference considering today's parallel systems are only capable of up to 5000 today.[4]

Some key design features:[2]

IBM's goal is to build the Blue Gene computer within five years and to immediately start working on modeling proteins with about 300 amino acids. The number of possible three-dimensional shapes for a protein is greater than the number of atoms in the universe, and the folding model will encompass more than a billion forces acting over a trillion different time steps. Although it is expected to be over 500 times faster than the fastest supercomputer today, Blue Gene will still take nearly one year of round-the-clock processing to solve the folding problem. [2]

Table C.1. File of PCs

1. **Embedded Memory** - the dynamic random access memory (DRAM) will be placed on board the processor reducing access time as well as power requirements
 2. **Minimalist Design** - the system will be built by replicating one chip 32,000 times
 3. **Multi-threading** - each of the 1 million processors can have 8 simultaneous threads in operation at a single moment in time
 4. **High Communications Bandwidth** - inter-processor communications will consist of six channels for each chip with a capacity of 2 gigabytes per second totaling to 300 terabytes per second. This is equivalent to everyone in the world (6.2 billion) operating four ISDN modems simultaneously and if the speed could be used to download the entire content of the internet—all 100 terabytes—Blue Gene could do it in less than a second
 5. **Small Footprint** - The final machine is hoped to have 64 interconnected racks, each rack containing eight boards, each board containing 64 chips, and each 20x20mm chip comprised of 32 processors. Each processor will operate at one gigaflop. Just one of the two-foot by two-foot boards will have the same raw computational power as the ASCI Blue Pacific supercomputer located at Lawrence Livermore National Laboratory.
- *Note: Any PC that has a number following its machine name indicates that there are x number of processors at that speed located on that machine.

Appendix D. Protein Structure Prediction Details

D.1 Background on the Protein Folding and Protein Structure Prediction Problems

This appendix contains background material on the protein folding and protein (or polypeptide) structure prediction problems, most of which has been presented in previous AFIT theses, particular Brinkman [4], Gates [41], and Deerman [25]. Section D.2 defines terminology in the biochemistry domain. Section D.3 describes the expensive experimental techniques used to determine the structure of proteins. Finally, Section D.4 examines various models used to predict the structure of polypeptides and proteins.

The protein folding problem Protein Structure Prediction (PSP) problem has been recognized as a National Grand Challenge problem in biochemistry and high-performance computing [15]. The challenge is to find a method to predict the three-dimensional topology of a protein based on the sequence of its components. A solution, which would provide knowledge about the function(s) of individual proteins, is also the first step toward solving the *inverse folding problem* (IPFP) [11, 82]. The inverse folding problem is to determine a sequence (possibly more than one) that fold to a specified three-dimensional structure.

The difference between the two problems is best characterized by the ability a solution to either would provide: a PFP solution would enable the *evaluation* of many proteins in a search for one with a specific property or function; an IPFP solution would provide a mechanism to *design* a protein with specified characteristics [11:25–26]. Possible applications include: pharmaceuticals with few or no side effects; energy conversion and storage capabilities (similar to photosynthesis); biological and chemical catalysts and regulators; angstrom scale information storage; and possible optical/chemical shielding from harmful radiation sources [11:25] [82:5] [114].

D.2 Introduction to Proteins and Associated Terminology

D.2.1 Chemical Underpinnings of Proteins.

D.2.1.1 Amino Acids. Twenty kinds of side chains varying in size, shape, charge, hydrogen-bonding capacity, and chemical reactivity are commonly found in pro-

Table D.1. Abbreviation of Amino Acids [139]

<u>Amino Acid</u>	<u>Three-letter Abbreviation</u>	<u>One- Letter Symbol</u>
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Asparagine or aspartic acid	Asx	B
Cysteine	Cys	C
Glutamine	Gln	Q
Glutamic acid	Glu	E
Glutamine or glutamic acid	Glx	Z
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

teins. In fact, proteins in all species, from bacteria to humans, are constructed from the same set of 20 amino acids (see Table D.1 for a complete list).

The simplest amino acid is glycine, which has just a hydrogen atom as its side chain. Alanine comes next, with a methyl group. Larger hydrocarbon side chains (three and four carbons long) are found in valine, leucine, and isoleucine. These larger aliphatic side chains are hydrophobic—that is, they have an aversion to water and like to cluster.

Proline also has an aliphatic side chain, but it differs from the other members of the set of 20 in that its side chain is bonded to both the nitrogen and a carbon atoms. The

resulting cyclic structure markedly influences protein architecture. Proline, often found in the bends of folded protein chains, is not averse to being exposed to water.

Three amino acids with aromatic side chains are part of the fundamental repertoire. Phenylalanine, as its name indicates, contains a phenyl ring attached to a methylene (-CH₂-) group. The aromatic ring of tyrosine contains a hydroxyl group, which makes tyrosine less hydrophobic than phenylalanine. Moreover, this hydroxyl group is reactive, in contrast with the rather inert side chains of the other amino acids discussed thus far. Tryptophan has an indole ring joined to a methylene group; this side chain contains a nitrogen atom in addition to carbon and hydrogen atoms. Phenylalanine and tryptophan are highly hydrophobic. The aromatic rings of phenylalanine, tryptophan, and tyrosine contain de-localized pi (p) electron clouds that enable them to interact with other p systems and to transfer electrons.

A sulfur atom is present in the side chains of two amino acids cysteine and methionine and both are hydrophobic. The sulfhydryl group of cysteine is highly reactive.

Two amino acids, serine and threonine, contain aliphatic hydroxyl groups. Serine can be thought of as a hydroxylated version of alanine. The hydroxyl groups on serine and threonine make them much more hydrophilic (water loving) and reactive than alanine and valine. Threonine, like isoleucine, contains two centers of asymmetry. All other amino acids in the basic set of 20, except for glycine, contain a single asymmetric center (the α carbon atom). Glycine is unique in being optically inactive.

We now turn to amino acids with very polar side chains, which render them highly hydrophilic. Lysine and arginine are positively charged at neutral pH. Histidine can be uncharged or positively charged, depending on its local environment. Two other amino acids contain acidic side chains and are known as aspartate and glutamate to emphasize that their side chains are nearly always negatively charged at physiological pH. Uncharged derivatives of glutamate and aspartate are glutamine and asparagine, which contain a terminal amide group in place of a carboxylate. [139]

D.2.2 Levels of Structure in the Protein Architecture. In nature proteins are found usually found in a single geometrical formation-it is possible for the protein to

Table D.2. Levels of Protein Structures [139]

Structure	Remarks
Primary	Is the amino acid sequence and the location of disulfides (if any) Is the complete description of the covalent connections of a protein
Secondary	Refers to the spatial arrangement (geometrical) of amino acids residues that are near one another in the linear sequence Some of these steric relationships are of a regular kind, giving rise to a periodic structure α -helix, β -pleated sheet, and collagen helix are elements of this structure
Super-secondary	Refers to clusters of secondary structure An example: a β strand separated from another β strand by an α -helix is found in many proteins (known as $\beta\alpha\beta$ unit) It is advantageous to regard this structure as an intermediary between the secondary and tertiary structures, as it is found that some polypeptide chains fold into two or more compact regions that may be joined by a flexible segment of polypeptide chain
Tertiary	Refers to the spatial arrangement (geometrical) of amino acids residues that are far apart in the linear sequence The dividing line between the secondary/tertiary is a matter of choice Proteins containing more than one polypeptide chain exhibit an additional level of structural organization and each polypeptide chain in this case is known as a sub-unit
Quaternary	Refers to the spatial arrangement of sub-units (mentioned above) and the nature of their contacts
Domains	In the super-secondary structure section it was discussed that some polypeptide chains form regions These regions (compact globular units) are known as domains and range in size from about 100 to 400 amino acid residues In some instances domains have been known to resemble each other in the same molecular construct which suggests that they arose by duplication of a primordial gene An important principal has emerged from analyses of genes and proteins in higher eucaryotes: protein domains are often encoded by distinct parts of genes called exons

vibrate slightly as a result of the interaction between the protein and thermal noise. This geometrical shape of a protein is directly related to the overall energy the protein has and it is believed that a protein folds to a geometrical shape that is associated with the overall minimal energy value.

When dealing with proteins and their physical structure, it is often beneficial to discuss levels of the structure. The following table provides the structure and associated remarks. [139]

Table D.3. ϕ, ψ Pairs of Common Secondary Structures

Secondary Structure	$\phi(\phi)$	$\psi(\psi)$
α helix (right handed)	-57	-47
α helix (left handed)	57	47
3_{10} helix (right handed)	-49	-26
Antiparallel β sheet	-139	235
Parallel β sheet	-119	113
Collagen helix	-51	153
Type II turn (second residue)	-60	120
Type II turn (third residue)	90	0
Fully extended chain	-180	-180

D.2.2.1 Known Secondary Structures of Proteins.

D.2.2.2 A Closer Look at Tertiary Structures in Proteins.

The three-dimensional structure of a protein is the major determinant of its function. This three-dimensional shape is called the *tertiary structure* or *conformation* of the protein. Proteins assume their *native* conformation, which is unique and compact, in their natural biological environment (typically in aqueous solution, at neutral pH and 20–40° C) [11, 82]. A protein in its native conformation is only slightly more stable than the various conformations with marginally higher energies. Normally, there is only a 10 kcal/mol energy difference between the completely folded and unfolded conformations. This single fact is responsible for the major difficulty of the protein folding problem [11:24–25] [82:2–4] [81:50].

There are two principle coordinate systems used to identify the position of the atoms in a molecule. The Cartesian coordinate system uses a three dimensional coordinate (x_i, y_i, z_i) , $1 \leq i \leq n$, where n is the number of atoms in the molecule. An arbitrary atom, usually C_{α_1} is assigned to the origin. This system is most useful to compute the distance, $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$, between two atoms. With this system each molecule has $3n$ degrees of freedom.

Internal coordinates is the other coordinate system. The dihedral angle approach defines position of all atoms in a protein from the position of one atom (usually at the origin), the *bond length* of each covalently bonded pair of atoms, the *bond angle* formed by each triplet of bonded atoms, and the *dihedral angle* formed by each bonded group of four atoms (see Figures D.1 - D.3). Given this set of parameters, every protein has $3n - 6$ degrees of freedom where n is the number of atoms. However, the bonds and bond angles are relatively rigid, therefore the independent dihedral angles are left as the only dominant factor to determine the tertiary structure of a protein and the degrees of freedom are reduced by a factor of approximately 2/3 [11:26] [81:50].

Each amino acid contains a Phi-Dihedral Angle ϕ , Psi-Dihedral Angle ψ , and Omega-Dihedral Angle ω dihedral angles and zero or more i th Chi Dihedral Angle χ_i dihedral angles as shown in Figure 2.2.

If we discretize the domain of the dihedral angles so that there are d possible values, then the size of the search space is given by d^N where N is the number of independently

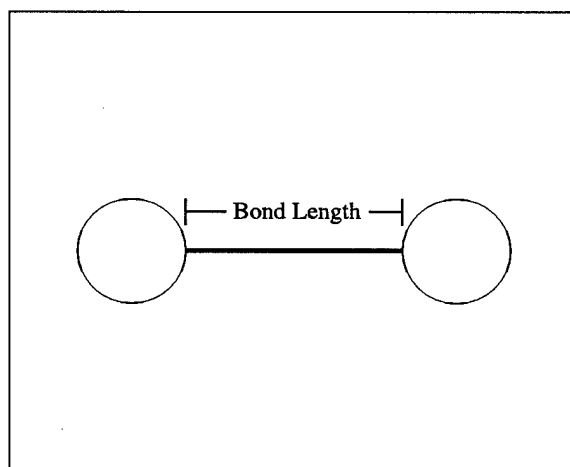


Figure D.1. Protein Bond Length

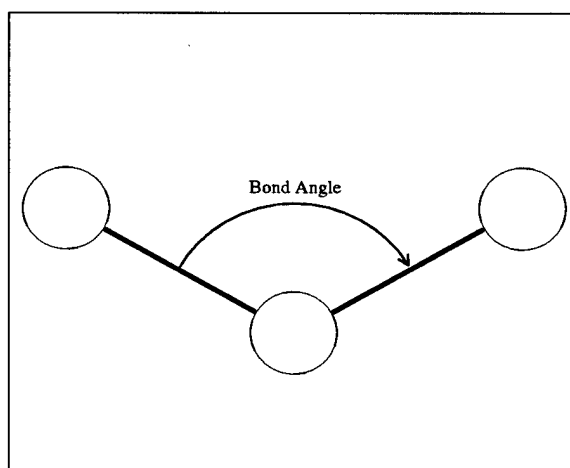


Figure D.2. Protein Bond Angle

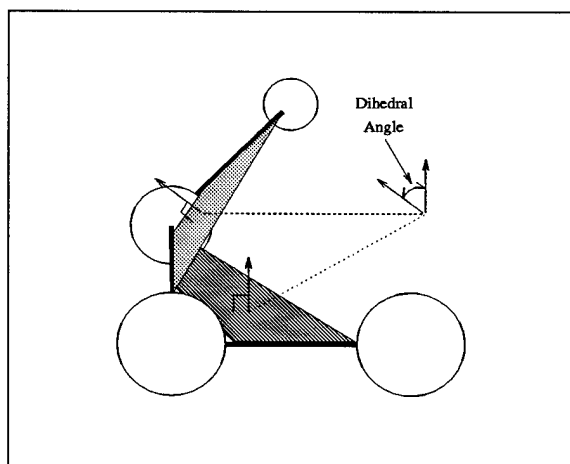


Figure D.3. Protein Dihedral Angle

variable dihedral angles. Given a very coarse 20° discretization of the $0 - 360^\circ$ dihedral angle domain and a small protein with 24 independently variable dihedral angles, the search space contains $18^{24} \approx 1.3 \times 10^{30}$ conformations. Table D.4 shows the time required to enumerate the search space on current and envisioned high performance computers (under the optimistic assumption of one evaluation per clock cycle)[137:7]! (Giga-, Tera-, and Peta-FLOP computers perform 10^9 , 10^{12} , and 10^{15} floating point operations per second, respectively) Therefore, if we hope to find the single native conformation of a protein, we must have access to efficient search algorithms that severely prune the search space.

Table D.4. Enumeration Time of a 1.3×10^{30} Search Space at One Solution per Clock Cycle

Computer Speed	Execution Time (years)
1 GigaFLOP	≈ 41 trillion
1 TeraFLOP	≈ 41 billion
1 PetaFLOP	≈ 41 million

D.2.3 Forces at Play. Weak, non-covalent forces play a key role in the folding of proteins into intricate three-dimensional forms. The three fundamental non-covalent bonds are electrostatic bonds, hydrogen bonds, and van der Waals bonds. They differ in geometry, strength, and specificity. Furthermore, these bonds are profoundly affected in different ways by the presence of water. The following table identifies specific characteristics of each. [139]

D.2.4 Conformation of a Protein. The PFP predicts the path taken by a protein (or polypeptide) transitioning from an unfolded (or denatured) state to its folded (or native) state. The elementary form of the folding reaction is deceptively simple (see Figure D.4). A solution to the protein-folding problem has eluded researchers for more than 30 years. [81] Tertiary Structure Prediction (TSP) is a related problem. The essence of the TSP is to predict the compact, three-dimensional shape of a protein, as it would exist in nature. This native conformation corresponds to the native state in the PFP. The native conformation determines the protein's biological functions.

Table D.5. Abbreviation of Amino Acids [139]

Non-covalent Bond	Characteristics
Electrostatic	<p>Charged group of substrate can attract oppositely charged group on an enzyme</p> <p>Force is derived by coulomb's law: $F = q_1q_2/r^2D$ in which the two "q's" are the charges of the two groups, r is the distance between them, and D is the dielectric constant (when D=1 in a vacuum, when D=80, in water)</p> <p>Also known as ionic bond, salt linkage, salt bridge, or ion pair</p> <p>Distance between oppositely charged groups in an optimal electrostatic attraction is 2.8 Å</p>
Hydrogen	<p>Can be formed between uncharged molecules as well as charged ones</p> <p>The hydrogen atom is shared by two other atoms (donor/acceptor)</p> <p>Donor is tightly linked with the hydrogen atom</p> <p>Acceptor has partial negative charge</p> <p>Hydrogen bond is can be considered as an intermediate in the transfer of a proton from an acid to a base</p> <p>Bond energies range from 3 to 7 kcal/mol</p> <p>Stronger than van der Waals bonds but much weaker than covalent bonds</p> <p>Highly directional—in fact the strongest hydrogen bonds are those in which the donor, hydrogen, and acceptor atoms are co-linear</p> <p>The alpha-helix is stabilized by hydrogen bonds between amide (—NH) and carbonyl (—CO) groups</p>
Van der Waals	<p>Nonspecific attractive force that comes into play whenever two atoms are within 3 to 4 Å</p> <p>Though weaker and less specific, they nonetheless play a significant role in the structure of a molecular construct</p> <p>The resulting attraction between a pair of atoms increases as they come closer, until the Van der Waals constant distance separates them (see figure). At a shorter distance, very strong repulsive forces become dominant because the outer electron clouds overlap.</p> <p>Van der Waals bond energy of a pair of atoms is about 1kcal/mol which is considerably weaker than hydrogen or electrostatic bonds and just above the threshold of thermal energy at room temperature (0.6 kcal/mol)</p> <p>A single van der Waals interaction counts for very little; however, it becomes significant when numerous atoms in one of a pair of molecules can simultaneously come close to many atoms of the other</p> <p>Though there is no specificity in a single van der Waals interaction, specificity arises when there is an opportunity to make a large number of van der Waals bonds simultaneously.</p> <p>Repulsion between atoms closer than the van der Waals contact distance are just as important as attractions for establishing specificity</p>

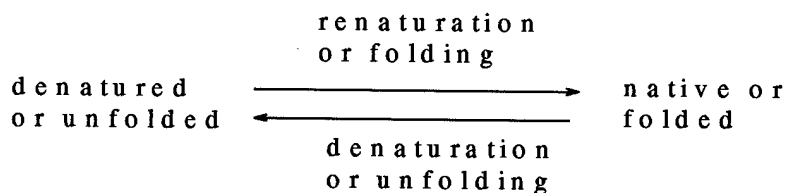


Figure D.4. The Folding Problem

D.3 Experimental Tertiary Structure Determination

Taking into account the preceeding sections, we now look at how can we apply the knowledge of atoms, amino acids, secondary structures, etc into consideration in determining Tertiary Structures of a given protein. In comparison with the number of known protein sequences, the number of known native conformations at high resolution is extremely small (less than 4000). The tertiary structures of these proteins have been determined experimentally using either X-ray crystallography or nuclear-magnetic-resonance (NMR) spectroscopy. These techniques are somewhat inadequate for the task as they take several months to obtain results for a single protein [11:25][82:5]. Thus, the quest is on for a method to accurately predict the structure without actually observing it. Even if this is not accomplished, a method that predicates the position of the backbone atoms can reduce the time required for experimentally determination to less than a year.

D.3.1 X-ray crystallography. X-ray crystallography is an experimental technique that exploits the fact that X-rays are diffracted by crystals. It is not an imaging technique. X-rays have the proper wavelength (in the ngstrm range, $\tilde{10}$ -8 cm) to be scattered by the electron cloud of an atom of comparable size. Based on the diffraction pattern obtained from X-ray scattering off the periodic assembly of molecules or atoms in the crystal, the electron density can be reconstructed. Additional phase information must be extracted either from the diffraction data or from supplementing diffraction experiments to complete the reconstruction (the phase problem in crystallography). A model is then progressively built into the experimental electron density, refined against the data and the result is a quite accurate molecular structure. [124]

For many, the X-ray crystallography is better than NMR because the knowledge of accurate molecular structures is a prerequisite for rational drug design and for structure based functional studies to aid the development of effective therapeutic agents and drugs. Crystallography can reliably provide the answer to many structure related questions, from global folds to atomic details of bonding. In contrast to NMR, which is an indirect spectroscopic method, no size limitation exists for the molecule or complex to be studied. The price for the high accuracy of crystallographic structures is that a good crystal must

be found, and that only limited information about the molecule's dynamic behavior is available from one single diffraction experiment. [124]

D.3.2 Nuclear Magnetic Resonance Spectroscopy. Nuclear magnetic resonance, or NMR as it is abbreviated by scientists, is a phenomenon which occurs when the nuclei of certain atoms are immersed in a static magnetic field and exposed to a second oscillating magnetic field. Some nuclei experience this phenomenon, and others do not, dependent upon whether they possess a property called spin. Spin is a fundamental property of nature like electrical charge or mass. Spin comes in multiples of $1/2$ and can be + or -. Protons, electrons, and neutrons possess spin. Individual unpaired electrons, protons, and neutrons each possesses a spin of $1/2$. [64]

Spectroscopy is the study of the interaction of electromagnetic radiation with matter. Nuclear magnetic resonance spectroscopy is the use of the NMR phenomenon to study physical, chemical, and biological properties of matter. As a consequence, NMR spectroscopy finds applications in several areas of science. NMR spectroscopy is routinely used by chemists to study chemical structure using simple one-dimensional techniques. Two-dimensional techniques are used to determine the structure of more complicated molecules. These techniques are replacing x-ray crystallography for the determination of protein structure. Time domain NMR spectroscopic techniques are used to probe molecular dynamics in solutions. Solid state NMR spectroscopy is used to determine the molecular structure of solids. Other scientists have developed NMR methods of measuring diffusion coefficients. [64]

The versatility of NMR makes it pervasive in the sciences. Scientists and students are discovering that knowledge of the science and technology of NMR is essential for applying, as well as developing, new applications for it. Unfortunately many of the dynamic concepts of NMR spectroscopy are difficult for the novice to understand when static diagrams in hard copy texts are used. The chapters in this hypertext book on NMR are designed in such a way to incorporate both static and dynamic figures with hypertext. This book presents a comprehensive picture of the basic principles necessary to begin using NMR

spectroscopy, and it will provide you with an understanding of the principles of NMR from the microscopic, macroscopic, and system perspectives. [64]

D.4 Tertiary Structure Prediction (PFP)

To reduce the gap between the number of known protein sequences and native conformations, we need to be able to reliably predict the tertiary structure of proteins in a reasonable amount of time. *Exact* versions of the classical methods discussed next are theoretically capable of finding the native tertiary structure of any protein. In practice, the computational cost of the calculations prohibits the use of these exact methods. The classical methods that are computationally viable are typically relaxed formulations that ignore the high-order interaction terms. The applicability of the other prediction methods discussed below is severely limited.

D.5 Classical Prediction Methods

Molecular dynamics is a technique that attempts to simulate the protein folding process. The protein is treated as an N-body simulation and Newton's motion equations are solved to determine the location of all the atoms at discrete points in time. Molecular dynamics faces two major difficulties in its attempt to fold proteins. First, the number of atoms that must be simulated is very large:

1. Small proteins contain hundreds of atoms.
2. Larger proteins can be composed of several ten-thousands of atoms.
3. Thousands of atoms must be added to simulate the surrounding solution.

Second, the thermal oscillations of bonded atoms have a period between 10^{-14} – 10^{-13} seconds. Simulation time steps in the femtosecond (10^{-15} sec) range are required to accurately account for these harmonics. These two factors have limited molecular dynamics simulations to less than a few nanoseconds (10^{-9} sec), even on today's fastest supercomputers. That time-frame is ten orders of magnitude too short to simulate the folding process of most proteins [11:27][82:6–7]. Using an *extended-atom representation* is one method that can greatly reduce the impact of these two problems. The extended-atom representation

combines hydrogen atoms with the heavier atoms they are bonded to. This representation generally halves the number of “atoms” in the problem and allows the size of the simulation time steps to be increased [7:189].

The *energy minimization* approach assumes that proteins, like other physical systems, assume that state which minimizes total energy in the system (however, this assumption is not universally accepted [67]). There are three types of energy minimization methods that differ by their time complexity and the accuracy of their calculations. *Ab initio* methods calculate the energy exactly. *Semi-empirical* methods eliminate the non-dominating interaction integrals from the calculation. *Force-field* methods simply account for the pairwise interactions between atoms with an appropriate parameterization that implicitly accounts for multi-particle interactions [82:6]. Table D.6 compares the time complexity of these three energy minimization models and gives example execution times for a moderately sized protein ($n = 1000$) assuming the individual component calculations take one nanosecond (10^{-9} sec).

Table D.6. Time Complexity of Energy Minimization Methods

Energy Calculation Method	Time Complexity	Time Estimate for $n = 1000$
<i>ab initio</i>	$\mathcal{O}(n^5)$	11.5 days
<i>semi-empirical</i>	$\mathcal{O}(n^4) - \mathcal{O}(n^3)$	17 min - 1 sec
<i>force-field</i>	$\mathcal{O}(n^2)$	1 msec

CHARMM and ECEPP are examples of force field energy models. The ECEPP/2 energy model is shown in Figure D.5. Its four terms represent the energy due to dihedral angle deformation, non-bonded interactions, electrostatic interactions, and hydrogen bond energy respectively. ECEPP is the most widely used energy model in PSP research. Our initial test molecule, Met-enkephalin, the accepted native structure has been identified by minimizations on the ECEPP model. It is a polypeptide specific energy model, thus it has limited utility for Wright Labs research into novel materials. The CHARMM [7] energy model is shown in Figure D.6. The five terms (which we denote E_B , E_A , E_D , E_N , $E_{N'}$) represent the energy due to bond stretching, bond angle deformation, dihedral angle deformation, non-bonded interactions, and 1-4 interactions, respectively. Other terms are available but are not implemented because their contributions are insignificant.

$$\begin{aligned}
E = & \sum_{(i,j,k,l) \in \mathcal{D}} \left(\frac{U_{ijkl}}{2} \right) (1 \pm \cos(n_{ijkl}\Theta_{ijkl})) + \\
& \sum_{(i,j) \in \mathcal{N}} \epsilon_{ij} \left[F_{ij} \left(\frac{r_0}{r_{ij}} \right)^{12} - 2.0 \left(\frac{r_0}{r_{ij}} \right)^6 \right] + \\
& \sum_{(i,j) \in \mathcal{N}} \left[332.0 \left(\frac{q_i q_j}{D r_{ij}} \right) \right] + \\
& \sum_{(i,j) \in \mathcal{H}} \epsilon_{ij} \left[\left(\frac{r_0}{r_{HX}} \right)^{12} - 2.0 \left(\frac{r_0}{r_{HX}} \right)^{10} \right]
\end{aligned} \tag{D.1}$$

Where

- \mathcal{D} is the set of 4-tuples defining ω and χ dihedrals,
- \mathcal{N} is the set of non-bonded atom pairs,
- \mathcal{H} is the set of hydrogen bonding atom pairs,
- r_{HX} is the donor-acceptor distance,
- r_{ij} is the distance between atoms i and j ,
- Θ_{ijkl} is the dihedral formed by atoms i, j, k , and l ,
- q_i is the partial atomic charges of atom i ,
- the U_{ijkl} 's, n_{ijkl} 's, F_{ij} 's, ϵ_{ij} 's, r_0 's, and D are empirically determined constants.

Figure D.5. ECEPP/2 Energy Model as Implemented by AGCT

$$\begin{aligned}
E = & \sum_{(i,j) \in \mathcal{B}} K_{r_{ij}} (r_{ij} - r_{eq})^2 + \\
& \sum_{(i,j,k) \in \mathcal{A}} K_{\Theta_{ijk}} (\Theta_{ijk} - \Theta_{eq})^2 + \\
& \sum_{(i,j,k,l) \in \mathcal{D}} K_{\Phi_{ijkl}} [1 + \cos(n_{ijkl}\Phi_{ijkl} - \gamma_{ijkl})] + \\
& \sum_{(i,j) \in \mathcal{N}} \left[\left(\frac{A_{ij}}{r_{ij}} \right)^{12} - \left(\frac{B_{ij}}{r_{ij}} \right)^6 + \frac{q_i q_j}{4\pi\epsilon r_{ij}} \right] + \\
& \frac{1}{2} \sum_{(i,j) \in \mathcal{N}'} \left[\left(\frac{A_{ij}}{r_{ij}} \right)^{12} - \left(\frac{B_{ij}}{r_{ij}} \right)^6 + \frac{q_i q_j}{4\pi\epsilon r_{ij}} \right]
\end{aligned}
\tag{D.2}$$

Where

- \mathcal{B} is the set of bonded atom pairs,
- \mathcal{A} is the set of atom triples defining bond angles,
- \mathcal{D} is the set of atom 4-tuples defining dihedral angles,
- \mathcal{N} is the set of non-bonded atom pairs,
- \mathcal{N}' is the set of 1-4 interaction pairs,
- r_{ij} is the distance between atoms i and j ,
- Θ_{ijk} is the angle formed by atoms i, j , and k ,
- Φ_{ijkl} is the dihedral angle formed by atoms i, j, k , and l ,
- q_i is the partial atomic charges of atom i ,
- the $K_{r_{ij}}$'s, r_{eq} 's, $K_{\Theta_{ijk}}$'s, Θ_{eq} 's, $K_{\Phi_{ijkl}}$'s, γ_{ijkl} 's, A_{ij} 's, B_{ij} 's, and ϵ are empirically determined constants (taken from the QUANTA parameter files).

Figure D.6. CHARMM Energy Model as Implemented by AGCT

D.6 Other Prediction Methods

Structure prediction by *homology* attempts to align the sequence of a protein with an unknown tertiary structure with one whose native conformation is known [82]. It has been observed that if the sequences are similar, then the conformations are frequently nearly identical. An extension of homology, called *sequence-structure alignment*, builds a partial monotonic mapping directly from the sequence of the unknown protein to the known tertiary structure of the similar protein. The differences between the two structures are usually surface characteristics built upon the same core structure. Both of these methods are severely limited by our tiny database of currently known protein structures. They are also incapable of predicting the native conformation of proteins with novel structures [82:7–9].

Simplification techniques are used as methods to reduce the conformational search space to a size that will to today's algorithmic search strategies [11]. *Lattice* models reduce three-dimensional space to a structured grid, where atoms can only be placed on the grid points. The grid is designed to accommodate the typical connections observed in real proteins. In such a discrete search space, the protein can be modeled as a walk on the lattice points, thus allowing for classical search techniques. The disadvantage is an obvious loss of fidelity by attempting to represent a continuous domain with a discrete approximation. Further simplifications have been obtained by reducing or eliminating the explicit representation of side-chains [82:9–10]. Dropping the side-chains will reduce the number of variables by up to one half. The optimal "fold" can be defined with just the backbone representation. But the minimal energy conformation must also consider contributions from the side chain.

D.7 Summary

The determination of the tertiary structure of proteins is a major challenge in biochemistry. Experimental techniques are considered accurate but time consuming, and are incapable of keeping pace with the number of protein sequences being discovered. Prediction techniques are hampered by the size of the conformational search space and the time complexity of calculating energy or solving motion equations. However, classical

prediction methods, combined with novel search and optimization algorithms, show great potential for both a solution to the protein folding problem and a better understanding of the underlying behavior and operation of biological systems. This thesis effort considers the application of genetic algorithms using energy minimization as one such combination for solving the protein folding problem.

Appendix E. Load Balancing pfmGA PSP Software

E.1 Introduction

AFIT's efforts in parallelizing a number of different software programs have met with much success. In fact there have been a number of theses that have been devoted exclusively to parallel implementations of various software packages that have been used to solve a number of scientific and engineering problems [9, 41, 88, 95, 96, 69, 25, 43, 4].

All the work, with the exception of Bohn, et.al. [9], concentrated on various strategies in which to implement parallel algorithms. Their intent was to improve the effectiveness of the algorithm with respect to finding a better solution or answer to the problem at hand. Bohn et. al, took another approach and attempted to provide a method in which one could analyze a heterogeneous computing environment a priori and using the derived information to load balance the algorithm to ensure each node had an appropriate amount of workload to process. Bohn et. al conducted there work on AFIT's Pile of PCs and the reader is encouraged to review there work and Appendix B for additional details concerning available parallel computing platforms for AFIT graduate research.

Our thesis effort in this area is similar to that of Bohn's except it is our intent to accomplish the same thing but with a much simpler methodology. During our initial look at trying to improve processor utilization with respect to the fmGA and the PSP problem the four potential methods were proposed and are discussed in the following section.

E.2 Improving Processor Utilization

Presently, AFIT's pfmGA PSP software works on any parallel (homogeneous-heterogeneous) platform; however, on a heterogeneous platform the slowest processor limits the performance. The pfmGA PSP software uses synchronous communication calls to conduct inter-communication between processors. This forces faster processors to wait idly for the slower machines to reach the same place in the code. The pfmGA PSP software has a number of these synchronous communication calls and thus the overall performance of the software is hindered.

Below are four ways in which to optimize performance in a heterogeneous environment:

E.2.1 Asynchronous Load Balancing. Code permitting, replace synchronous communication calls with asynchronous communication calls

This is not a viable option as there are too many inter-dependencies (at least for some cases of the way the pfmGA PSP software is being utilized) between processors at specific points in the algorithm. For example, the pfmGA PSP software consists of three phases (Initialization, Primordial, and Juxtapositional) and between each phase inter-processor communications exists.

E.2.2 Systems Performance Analysis and Load Balancing. The idea here is to determine actual performance of each individual processor in the heterogeneous cluster and assigning an associated speedup variable to each machine with a 1 being assigned to the overall slowest machine.

This method requires a detailed preliminary investigative effort on the part of the programmer to encapsulate individual speedup based on the slowest machine. As shown by Bohn & Lamont [9] symmetric Load Balancing on the AFIT's Pile of PCs was not a trivial effort. While we could readily use the results of their effort, we felt the amount of effort required doing the background investigating offsets the gain, especially when we considered the third and fourth options.

E.2.3 Dynamic Comparison Performance Load Balancing. This type of Load Balancing would alter the code at those places where synchronous communications is taking place to determine how much idle time is available to the faster processors

To accomplish this the following must occur

1. Start timer before entering synchronous communications
2. End timer after communications has taken place
3. Difference is equal to the amount of time waited by the faster processor plus communication costs

```
For (i = time_available_to_do_other_work; i > 0; i = i - time_to_do_extra_work)
    Do extra work;

Where for this example:
    time_available_to_do_other_work = 70
    time_to_do_extra_work = 11
```

Figure E.1. Pseudo-code for Extra Work Loop

4. Remove communications costs by taking the difference between the start and stop timer on the slowest machine and subtracting that from the faster machines start-stop time difference

This method allows one to dynamically alter the amount of "extra" work performed by the faster machines by taking differences in times of the faster machines and slowest machine and using that value as termination criteria for the control loop. For instance suppose the fastest machine reaches a critical synchronization point in the code 70 seconds before the slowest machine. Furthermore, suppose that a piece of code that can be executed independent of the other code in 11 seconds. An "extra work" control structure could be set up just before the critical synchronization point such that for this example, the extra work loop would be executed 7 times.

However, there is a problem with this in that for six loops the cost in time is 66 seconds, for the seventh loop, the slowest processor will have to wait 7 seconds. Other machines slower than this machine, but faster than the slowest machine having to wait somewhere between 1 and 7 seconds. The cost of having all slower machines having to wait on the faster machines to finish up their extra work becomes an issue as additional slower machines are added to the pool of heterogeneous machines.

To remedy this, the extra work loop's control mechanism should be changed to reflect the following:

Changing this code will result in only getting 6 iterations of the extra work loop; however, if one could choose the amount of work to be done such that the time it takes one iteration to complete approaches 0, then the # of iterations will approach infinity (i.e. wasting very little computational resource).

```
For (time_to_do_extra_work = time_available_to_do_other_work;  
    i > 0; i = i - time_to_do_extra_work)  
    Do extra work;
```

Figure E.2. Improved Extra Work Loop

E.2.4 System Performance Analysis/Dynamic Comparison Load Balancing. This strategy is a combination of the prior two strategies-with the second strategy being slightly modified to reduce the amount of analysis and goes one step beyond the third method in that it handles "some" of the spare time associated with the first run. Recall that for the third method, the amount of time waiting, or conducting, what will become the extra work must be calculated. In order to do this, the entire program must run at least once. During the first run, no performance improvement is realized with method three, hence a combination of 2 & 3 is proposed.

By incorporating a predetermined factor number-something simple should be used here as it is only "solely" affects the first run, as subsequent runs will use both the speedup factor as well as the timer method and the timer method will ensure all spare time is used. In essence, the timer method is used to augment the speedup factor to essentially fill in the remaining gap of available processor time. A simple speedup factor could be based merely on a percentage-this value should be no greater than 75%-difference associated with the slowest processor speed and all others. For example, if the slowest machine is 133MHz and the fastest is 600MHz, then a speedup value of the faster machine would be equal to $(600/133)*.75$ or 3.38. In general the decimal places should be dropped and the floor of the value taken. The slowest machine will always have a speedup factor value of 1.

The algorithm can take advantage of the speedup factor by carefully incorporating it into the algorithm where a loop takes place. For instance, if you have a loop that is doing some operation 5 times, then the slowest machine would do that loop 5 times, while the faster machine would conduct it 15 times ($3*5$).

To determine individual speedup factors, additional information is required at the start. Each node will have determined who it is by calling a system routine. Once called

that information is used in a case statement that identifies each machine with an associated processor speed.

Further research should be accomplished here to improve the overall performance of pfmGA and other genetic algorithm in a heterogeneous environment.

E.3 Conclusions

The third method was selected and implemented on the pfmGA PSP software with great success (see the Chapter on Experimental Results).

Appendix F. Determining Population Size

F.1 Introduction

The population size is an important aspect to any genetic algorithm and its choice is crucial to a successful implementation (success defined through performance and effectiveness). Goldberg, et. al. Analyzed the population sizing problem in (35).

The formula derived is based on the assumptions and is therefore an upper bound for choosing a population size. Gas using population sizes based on their formula (equation #) should converge somewhere between $O(l \log l)$ and $O(l^2 \log^3 l)$ function evaluations depending on the selection scheme used.

$$n = 2c \left(\frac{\sigma_{rms}^2 (m-1)}{d^2} \right) \chi^k \quad (F.1)$$

Where n = population size c is a parameterized constant of the confidence factor you want to enforce (a)

$\sigma_{rms}^2 (m-1)$ is an estimate of the variance of the average order- k schema ($m=1/k$)

d is the signal difference we wish to detect

c is the cardinality of the encoding alphabet

and k is the estimated order of deception in the problem

Gates in his thesis (Predicting Protein Structure Using Parallel Genetic Algorithms) [41] calculated the theoretical population sizes for Worst, Measured, and Best cases and derived the values contained in Table F.1 (refer to his thesis to obtain his "conservative" assumptions used to derive the resulting population sizes).

Table F.1. Gate's Predicted Population Sizes to Minimize the [Met]-Enkephalin based on Goldberg's Population Sizing Equations

Variance Assumption	Calculation Parameters l k χ c d ρ^2 rms	Population Size
Worst Case	240 5 2 6 0.1 1.65 X 1023	2.98 X 1029
Measured Case	240 5 2 6 0.1 9.20 X 1018	1.66 X 1025
Best Case	240 5 2 6 0.1 8.95 X 10-48	1.62 X 10-41

Unfortunately, as Gates points out, the first two cases result in extremely large population size requirements that for all practical purposes are impossible to implement and the best case population size is trivial as it requires no population size (can't have a negative population size). To date, no theoretical work can provide a useful population size estimate for the pfmGA PSP software.

Gates derived the current method of determining the population size for the pfmGA PSP software through the use of Merkle's work [89]. The essence of their combined effort was to determine a reasonable population size that could be used. This population size was through an order-preserving transformation on the fitness values. Since this order-preserving transformation also reduced the variance of the fitness function Gates concluded that the population size required by the pfmGA PSP would be less than the theoretically derived population sizes. Gates repeatedly applied the order-preserving transformation to obtain a fixed population size of 4512 after 58 iterations.

Appendix G. Evolutionary Computation

G.1 Introduction

The field of Evolutionary Computation is really the coming together of Evolutionary Strategies, Evolutionary Programming, Genetic Programming, and Genetic Algorithms. The two Evolutionary subcategories concentrate more on mutation as the main operator, while The Genetic subcategories concentrate more on crossover. In general though there is a very thin line as to what the two classes of evolutionary algorithms a given algorithm will fall into.

This Appendix provides additional information about Evolutionary algorithms in general and genetic algorithms in specific. The appendix is ordered from the general to the specific. In addition the chapter discusses briefly the schema theory and its incorporation into genetic algorithms.

G.1.1 Brief History of Evolutionary Algorithms. Evolutionary models based on natural selection and genetic theory started appearing during the late 1950s and early 1960s. The first working models were computer simulations of genetic and biological systems. The idea of using algorithms that model natural evolution as search and optimization techniques began in the late 1960s and 1970s [46:89–104]. The class of methods called evolutionary algorithms (EAs) is composed of three main categories based on that early work: Evolutionary Programming (EP), Genetic Algorithms (GAs), and *Evolutionstrategie* (Evolution Strategies, ESs) [1, 21] [46:104–106]. However, this taxonomy is not universally accepted. Many authors categorize only EP and ESs as Evolutionary Algorithms, and put GAs in a separate category by themselves [36:24]. Even though simulated annealing (SA) is based on thermodynamics, it's often associated with evolutionary algorithms because it uses a mutation operator [135:17]. However, SA fail other “test” for inclusion with EA's because it operates on a single candidate solution rather than a population. Also difficult to categorize are hybrid combinations, such as GAs with deterministic local search or GAs with novel data representations. Such algorithms are simply called EAs [99].

Fogel, Owens, and Walsh first proposed the technique known as evolutionary programming. Evolutionary programming tries to generate computational biological evolution

through a process that allows the survival of organisms that respond appropriately to a given environment. That is, it attempts to evolve a Finite State Machine (FSM). It has been applied to problems such as sequential symbol prediction and process control. EP usually operates on the components of abstractions such as finite state machines or programming languages [1] [35] [37].

Evolution strategies was conceived by Ingo Rechenberg and Hans-Paul Schwefel at the Technical University of Berlin while they were searching for optimal airfoil shapes. The original formulation consisted of a one-member population that was operated on by mutation only. The representation consisted of a pair of real-valued vectors ($V = \langle \mathbf{x}, \sigma \rangle$) where the first vector, \mathbf{x} , represents a solution and the second vector, σ , is a vector of standard deviations. The mutation operator creates a new individual at the t generation using a normal distribution with zero mean as follows:

$$\mathbf{x}^{t+1} = \mathbf{x}^t + N(0, \sigma) \quad (\text{G.1})$$

Various refinements have been made to this original design including population sizes greater than one, recombination operators, and dynamic changes to the vector σ [99:160–164][1].

G.2 Genetic Algorithms

Most the following material has appeared in previously AFIT theses, Brinkman [4], Dymek [28], Gates [41], and Merkle [88]. Section G.2.1 provides a historical context for evolutionary algorithms. Sections G.3 through G.5 present the theory and mechanics of simple GAs (SGAs), messy GAs (mGAs), and fast messy GAs (fmGAs) respectively. Finally, Section 2.6.2 describes techniques used to parallelize genetic algorithms.

Genetic algorithms (GAs) are a stochastic search technique loosely based on natural evolution and the Darwinian concept of “Survival of the Fittest” [46:1][63]. A generalized genetic algorithm consists of a *population* of *encoded* solutions that are manipulated by a set of *operators* and evaluated by some *fitness function* that determines which solutions survive into the next *generation*.

For our purposes, search and optimization techniques fall into two broad categories: deterministic (a combination of calculus-based and enumerative) and stochastic (random) methods [46:2]. Greedy algorithms, hill-climbing, calculus-based methods, and branch and bound tree/graph search techniques are all examples of deterministic approaches [3]. These methods have been successfully used to solve a wide variety of problems. However, there is an even greater number of large dimensional problems that are discontinuous, multi-modal, or NP-complete where deterministic methods are ineffective [46:3–6][35, 39]. The main short coming for deterministic methods is their requirement for some amount of problem specific knowledge to direct or limit the search. For example:

1. Hill-climbing algorithms are limited unimodal functions
2. Greedy algorithms assume optimal sub-solutions are *always* part of the optimal solution [3:80][76]
3. Calculus-based methods require continuity [80:167]
4. Branch and bound search techniques need problem specific heuristics/decision algorithms to limit the search space [39, 115]

A partial list of problem characteristics that can make deterministic search techniques unsuitable for a particular problem includes: multi-modal and/or discontinuous solution spaces, exponential search spaces (NP-complete problems), and limited domain knowledge (no heuristics). Problems that exhibit one or more of these characteristics are called *irregular* [79].

Stochastic search and optimization approaches (simulated annealing, evolutionary strategies, evolutionary programming, genetic algorithms, Monte-Carlo techniques) have been developed as an alternative to deterministic techniques [46, 99]. The only requirements for stochastic methods are a function that assigns fitness values to possible solutions and an encode/decode between the algorithm and problem spaces. Although these methods cannot guarantee the optimum solution, in general they can provide *good* solutions to a wide range of problems that may be irregular and/or exponentially too large dimensionally for deterministic methods [46:6–7][76]. Simulated annealing and Monte-Carlo techniques, in addition to GAs, are frequently applied to the PSP problem.

G.2.1 Origins of Genetic Algorithms. John Holland's work on adaptive systems is recognized as the fundamental beginning of genetic algorithms. Previous researchers had used computers to simulate evolutionary systems, and Fraser even tried to optimize a phenotype¹ function, but nobody before Holland recognized the role that nature's evolutionary process could play in search and optimization [46]. The embarkation point for all GA research is Holland's "*Adaptation in Natural and Artificial Systems*" [62] which established the mathematical basis for GAs, including the then unnamed *Schema Theorem* or *Fundamental Theorem of Genetic Algorithms*, discussed in Section G.3.3, and generalized schemes for reproduction, crossover, mutation, and inversion [46:89–92].

Three other names have come to be synonymous with GA research: Kenneth A. De Jong, David E. Goldberg, and John J. Grefenstette. De Jong's dissertation [22] put Holland's theory to the test and introduced GA infrastructure that is still in use today (a suite of test functions and performance measures). Although his dissertation focused on function optimization, most of his work to date concerns his broader interest in machine learning [133, 23, 24]. Goldberg began by applying genetic algorithms to machine learning and optimization problems. His most recent efforts include work on optimal GA population sizes and alternate GA paradigms (messy GAs and fast messy GAs) to combat deceptive problems [46:387–389] [50, 48]. Grefenstette is probably best known for his genetic algorithm implementation, GENESIS, which has been used as a basic GA workbench by many researchers, including those at AFIT [56]. He has also worked on optimal GA parameter sets and machine learning using genetic algorithms [54, 58, 55, 57].

G.3 Simple Genetic Algorithm (SGA)

Simple genetic algorithms are based on theories of natural genetics and therefore share some of the same terminology. Figure G.1 illustrates the following terms. A *string* or *chromosome* contains *genes* that encode a solution to a particular problem. A *locus* and *allele* are associated with the gene. The locus, or position of a gene generally determines

¹The term **phenotype** refers to the traits expressed by an individual, in this case the value returned by a function. Contrast this with **genotype** which refers to the traits that define the individual, for example the parameters of the function

problem association. A gene is given only one value or allele at a time from a set of values (the alleles) allowed for that gene. A bag of strings is called a *population*. A genetic algorithm *evolves* populations toward better solutions of the encoded problem by generations.

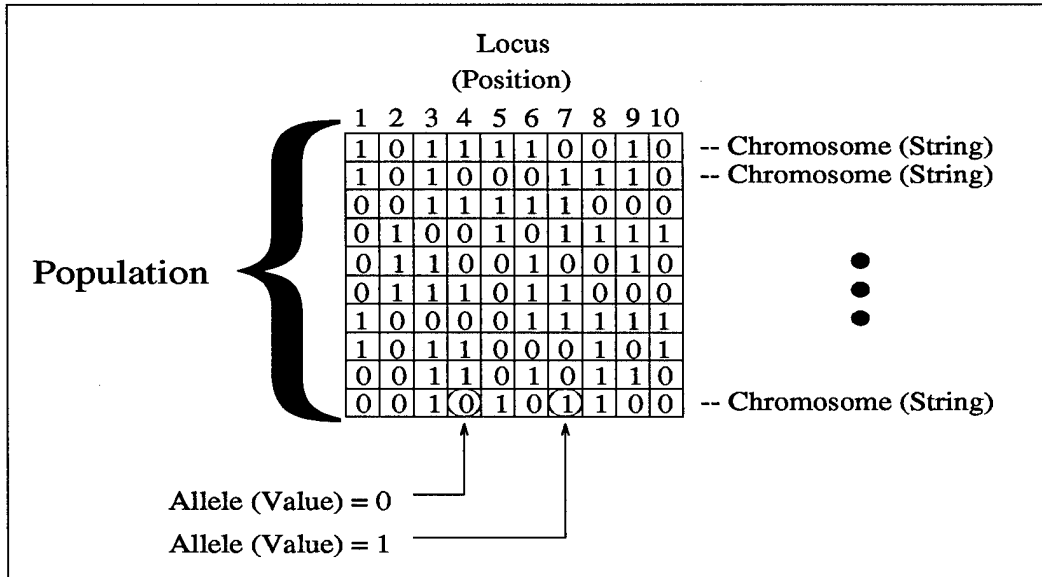


Figure G.1. Simple Genetic Algorithm Data Structures and Terminology

Most SGA implementations use strictly binary encodings of the problem parameters, usually in a form such that x_{min} corresponds to a string of all 0's, x_{max} corresponds to a string of all 1's and there is a linear mapping of all values between x_{min} and x_{max} . Some problems may benefit from the use of Gray code parameter encoding [46:101]. In Gray code, the encoding of successive integers differ by a single bit [78:40]. It improves a mutation operator's chance of making incremental improvements, thus increasing the exploitation of favorable portions of the search space. Higher cardinality encodings have also been investigated for certain other problems, most notably combinatoric problems [112, 136]. The problem encoding is a very important design decision in the formulation of a genetic algorithm to solve a specific problem. Binary offers the greatest domain independence while other representations are often more "natural" for, but limited to, a specific problem.

G.3.1 Simple Genetic Algorithm Operators. The three standard operators associated with simple genetic algorithms are *selection*, *crossover*, and *mutation* [46, 99]. Above-average individuals of a population are *selected* to become members of the next generation more often than below-average individuals. *Crossover* recombines pieces of solutions to test different combinations of existing solutions. In the absence of other operators, selection and crossover will eventually force a population of solutions to *converge* to a single solution [46:14] [52]. *Mutation* is an operator designed to encourage diversity in a population so that convergence occurs more slowly and more of the solution space can be explored.

Figures G.2 and G.3 illustrate the function of *single-point crossover* and *bitwise mutation* respectively on binary encoded strings that are ten bits long. Crossover operates on two strings, called the parents, to create two new strings, called the children. After two parents and a crossover point have been arbitrarily chosen, the bits after the crossover point are exchanged to create the children. Mutation is a unary operator that takes one string as input, arbitrarily chooses a bit position within the string, and changes the bit in that position to the opposite value. Many other crossover and mutation operators are possible, and indeed necessary, to exhibit different recombination characteristics and operate on alternate encodings [136, 145].

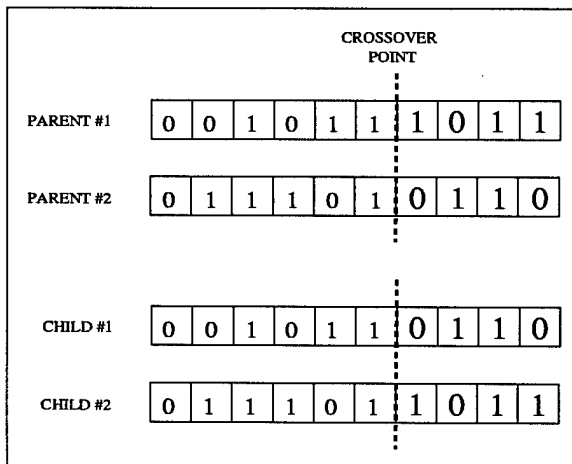


Figure G.2. Single-Point Crossover

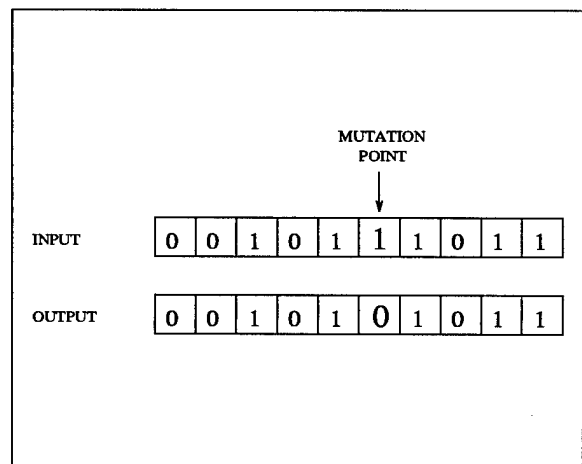


Figure G.3. Bitwise Mutation

Figure G.4 represents the operation of a *proportional* selection operator, called *roulette-wheel* selection, on two different populations of four strings each. Each string in the population is assigned a portion of the wheel proportional to the ratio of its fitness and the population average fitness. In the first case where the fitnesses are equal, each string is

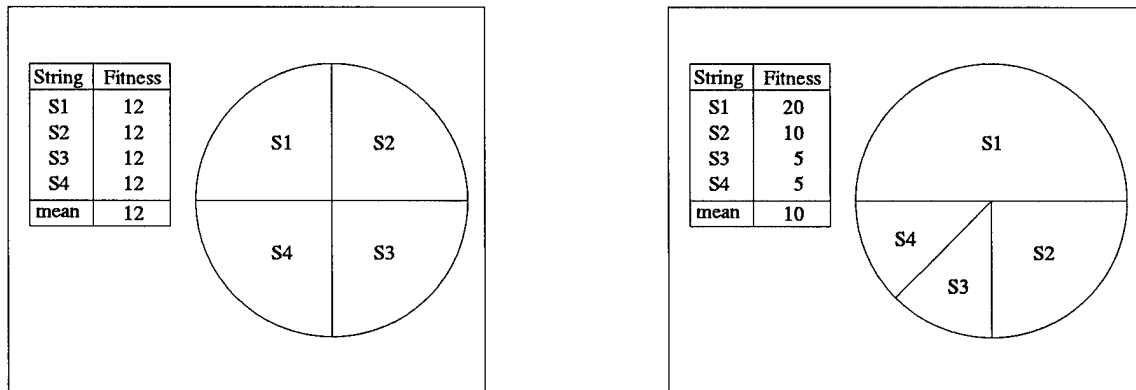


Figure G.4. Roulette Wheel Selection

given an equal share of the wheel (it is equi-likely that any of the four strings is selected for the next generation). In the second example, *S1* is twice as likely to be selected for the next generation as *S2*, which is twice as likely to be selected into the next generation as either *S3* or *S4*. As with crossover and mutation, many other selection operator variations are possible, each with its own characteristic effect on convergence. *Rank-based* and *tournament* are notable selection operators, and the *elitist* strategy is a modification that can be used with any selection operator [46, 135, 144].

The three operators (crossover, mutation, and selection) and an evaluation function are assembled according to the pseudo algorithm shown in Figure G.5 to create a simple genetic algorithm. Figure G.6 is a graphic representation.

G.3.2 Simple Genetic Algorithm Parameters. The most difficult part of genetic algorithms is selecting a parameter set that will generate the best performance (efficiency and effectiveness). Efficiency is a measure of the computer resources (cpu time, memory) required to obtain a solution. Effectiveness relates the solution quality of various algorithms. Both measures are relative to the specific problem being tackled and tradeoffs can be made between the two.

1. randomly generate initial population
 2. evaluate fitness of all population members
for $i = 1$ to the maximum number of generations and
not some other stopping condition
 3. perform selection
 4. perform crossover
 5. perform mutation
 6. evaluate fitness of all population members
- end loop

Figure G.5. Pseudo Algorithm for Simple GAs

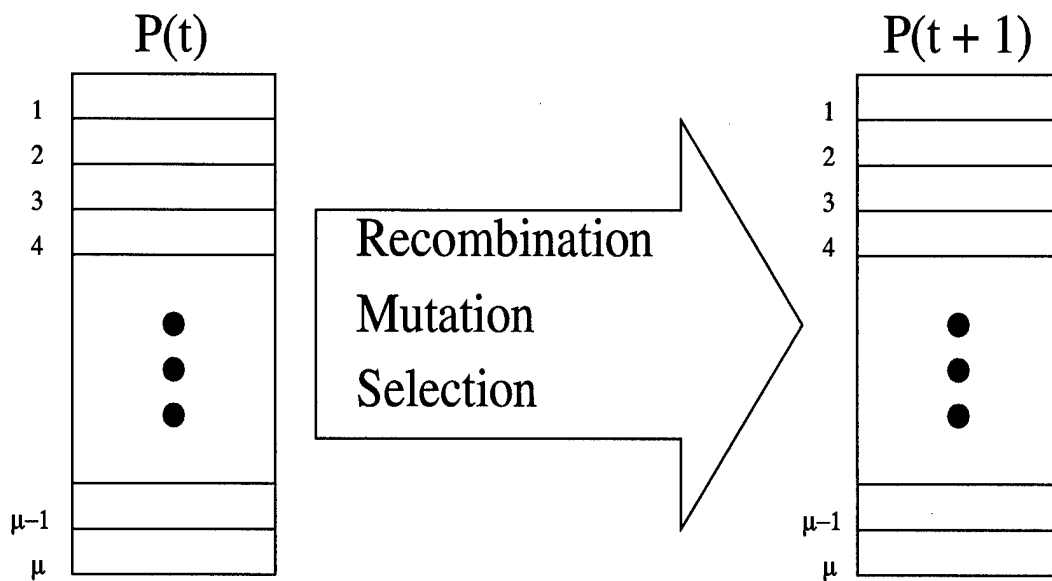


Figure G.6. Simple Evolutionary Algorithm (GA)

Some interrelated SGA parameters include: population size, crossover probability, and mutation probability. Theoretical analysis and empirical studies have been accomplished to formulate estimates of what each of these parameters should be to encourage a robust search that terminates with a near-optimal solution [47, 130]. While these two particular studies seem at odds with each other, their conclusions are based on entirely different measurements of GA progress. Schaffer's empirical study is aimed at maximizing *on-line performance* or progress toward optimal solutions (efficiency) without regard for the final solution. Goldberg's theoretical work makes conservative choices to estab-

lish confidence levels for the optimality of a final solution (effectiveness) and ignores the astronomical resource costs required to achieve it.

Many relationships have been observed between parameter settings and performance. Increasing the population size generally improves the final solution, however the increase in execution time becomes prohibitive [22, 45]. Population size has been shown to exhibit an inverse relationship with mutation rate and, to a lesser extent, crossover rate [130:55]. Although there is no evidence so far of any correlation between crossover and mutation probabilities it is generally accepted that using crossover without mutation is insufficient for a robust search [32, 76, 131]. However, it has been postulated (especially from the other branches of evolutionary algorithms) that mutation is the only necessary operator [36, 131]. Other researchers are examining the effects of changing parameter settings during GA execution, either on some predefined schedule or possibly by monitoring GA metrics during the run [19, 34].

G.3.3 Mathematical Theory of How (Why) Simple GAs Work. *Schemata* are templates that define sets of strings with the same values at certain string positions and are represented using an additional *don't care* symbol (*) [46:19,29]. For example, the schema *101 represents the set of strings {0101, 1101} and the schema 1*0*01 defines the set {100001, 100101, 110001, 110101}. The *defining length* ($\delta(H)$) and *order* ($o(H)$) are two values associated with a particular schema H . The defining length of a schema is a measure of the distance between the first and last fixed positions. The order of a schema is the number of positions with fixed values. Using the sample schemata from above $\delta(*101) = 4 - 2 = 2$, $o(*101) = 3$, $\delta(1*0*01) = 6 - 1 = 5$, and $o(1*0*01) = 4$.

The Schema Theorem, represented by

$$m(H, t + 1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l - 1} - o(H)p_m \right], \quad (\text{G.2})$$

establishes a lower bound on the number of representatives schema H will have in the next generation ($m(H, t + 1)$) based on the:

1. number of representatives in the current generation ($m(H, t)$),

2. fitness of schema H vs the population average fitness ($\frac{f(H)}{f}$),
3. string length, defining length, and probability that schema H will be destroyed by crossover ($p_c \frac{\delta(H)}{l-1}$), and
4. order and probability that schema H will be destroyed by mutation ($o(H)p_m$).

Although it would appear that genetic algorithms operate only on the specific strings in a population, it has been shown that many of the 2^l schemata in each string are processed simultaneously (*implicit parallelism* [62:71–72][46:40]). The Fundamental Theorem of Genetic Algorithms (Schema Theorem) states that all schemata will receive representation in the next generation proportional to the ratio of their fitness to the average fitness of the population. This representation is reduced by the amount of disruption that crossover and mutation can cause to a schema. More succinctly,

short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations [46:33].

G.3.3.1 Complexity Analysis. Using the standard SGA operators, the time complexity of selection, crossover, and mutation are generally $\mathcal{O}(nl)$ where n is the population size and l is the string length. Given a fixed number of generations, SGA execution time is $\mathcal{O}(nl)$ [26]. However, the time complexity of the fitness function for real-world problems (expressed as some function of the string length and problem space parameters) usually dominates the time to execute the genetic algorithm control sequence [104], therefore great care should be taken in its analysis and design.

It is easy to see that the space complexity of a simple genetic algorithm is also $\mathcal{O}(nl)$ because the population of solutions has to be stored. However, the space complexity can also be affected by the data structure requirements of specific problems. For example, a function that relies on a lookup table to evaluate solutions requires more space. If that space is related to the problem size in a way that make the lookup table grow faster than the population size and string length, then the problem space requirements dominate the space complexity of the entire GA algorithm.

G.4 Messy Genetic Algorithm (mGA)

Genetic algorithms are designed to take advantage of the *building block* theory [38, 46, 62]. The main idea is that small pieces of a solution which exhibit above average performance are combined to create larger pieces of above average quality, which are themselves recombined into larger pieces, and so forth.

Simple genetic algorithms suffer from the fact that the “pieces” that form the building blocks must be put next to each other explicitly in the fixed encoding or else they are more likely to be disrupted by crossover. This problem is magnified when competing schemata (schemata with different values at similar defining positions) define locally optimal solutions. *Deception* occurs when a locally optimal building blocks are selected instead of globally optimal ones. Messy genetic algorithms (mGAs) were designed to deal with these problems by encoding the string position (locus) along with its value (allele). This gives a messy genetic algorithm the ability to search for the “true” building blocks of the problem and create tighter *linkage* for those genes than a fixed position encoding would allow [51]. The mGA encoding scheme also allows *under-specified* and *over-specified* strings to exist in the population. Under-specified strings don’t have an allele defined for every locus and are evaluated with the aid of a locally optimal *competitive template* that supplies values for the unspecified genes. Over-specified strings contain multiple alleles specified at the same locus and are processed in a left-to-right fashion which sets the gene to the value encountered first. The desire to create and manipulate superior building blocks is the motivation behind messy genetic algorithms [50, 50, 49].

G.4.1 Messy Genetic Algorithm Operators. Messy GAs use variations of the same genetic operators used by simple GAs. In the few implementations of mGAs that exist [50, 51, 49, 88], tournament selection has been used instead of proportional or rank-based selection because of its desirable performance characteristics [50:50][33, 47]. The tournament selection operator also has a *thresholding* mechanism added to it which ensures that strings have a number of positions in common before competition is allowed [49:424–427]. Crossover is replaced by a combined *cut-and-splice* operator that works on variable length strings. As the names suggest, *cut* divides a string into two smaller pieces and

splice concatenates two strings to form a single, longer string. A mutation operator that can change a gene's value or its position has been described but unused in any mGA implementations [51:504].

Messy GAs employ a different initialization strategy compared to SGAs. The main processing loop of an mGA is composed of *primordial* and *juxtapositional* phases. During *partially enumerative initialization (PEI)*, exactly one copy of each possible building block of the specified size (k) is generated. Thus, the initial population size for a messy GA is generally quite large ($2^k \binom{l}{k}$) [51:420]. The primordial phase serves two basic purposes: enrich the population with above average building blocks and reduce the population to a size that can be efficiently and effectively processed by the juxtapositional phase. Tournament selection, the only active operator during the primordial phase, fills the population with above average building blocks, then periodically the population size is halved. No additional fitness evaluations are required during the primordial phase. The juxtapositional phase is most similar to the main processing loop of a simple GA [51:506]. Cut-and-splice and any other genetic operators are applied, fitness evaluations are performed on the newly created strings, and tournament selection bolsters the next generation with highly fit solutions. A pseudo algorithm for messy GAs is shown in Figure G.7.

1. perform partially enumerative initialization
 evaluate fitness of all population members
2. for $i = 1$ to the maximum number of primordial generations
 perform tournament selection
 if (a suitable number of generations have transpired) then
 reduce the population size
 end if
 end loop
3. for $i = 1$ to the maximum number of juxtapositional generations
 perform cut-and-splice
 perform other operators (currently not used)
 evaluate fitness of all population members
 perform tournament selection
 end loop

Figure G.7. Pseudo Algorithm for Messy GAs

G.4.2 Messy Genetic Algorithm Parameters. The major parameter settings associated with messy GAs are population size, cut-and-splice probabilities, and a schedule for reducing the population size. Initial population size can be calculated once the string length and block size have been determined. String length is simply a function of the encoding used, but block size is a problem dependent quantity that may be difficult to estimate. The final population size at the end of the primordial phase is even less quantifiable! The splice probability is consistently set to 1.0 with the following rationale: the primordial phase ends with a population of optimal building blocks which should only require assembly to form a complete string that is a near-optimal solution [50:25]. The chosen cut probability is scaled by the current length of a string so that longer strings are more likely to be cut than shorter strings. The schedule for reducing population size during the primordial phase typically allows for two or three generations of enrichment followed by cutting the population in half [51:505]. No theoretical or empirical work has been accomplished to provide any guidance for final primordial population size, cut probability, or population reduction schedules for messy GAs.

G.4.3 Mathematical Theory of How (Why) Messy GAs Work. The Schema Theorem (Equation G.2) is directly applicable to messy genetic algorithms. The rationale for messy genetic algorithms follows from the theorem's interpretation: "short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations." If the building blocks of a problem aren't encoded as short, low-order schemata then crossover and mutation will disrupt the formation of those building blocks.

For problems using a fixed encoding, where the identification of building blocks is prohibitive or impossible, Goldberg has calculated the *normalized expected defining length* ($\frac{\langle \delta \rangle}{l+1}$) for k -sized building blocks (Equation G.3). The normalized expected defining length is a measure of the mean length of the schemata that make up the building blocks of a randomly encoded problem. The interpretation is that an arbitrary encoding is highly unlikely to establish tight linkage for the building blocks of a problem [51:498–499].

$$\frac{\langle \delta \rangle}{l+1} = \frac{k-1}{k+1} \quad (\text{G.3})$$

Messy genetic algorithms take advantage of the Schema Theorem by searching for both the defining positions and gene values of the building blocks using PEI and the primordial phase. Then the juxtapositional phase of the messy GA starts with “short, low-order, above-average” schemata that are also “short, low-order, above-average” building blocks!

G.4.3.1 Complexity Analysis. Because of the partially enumerative initialization (PEI), the time complexity of messy GAs is $\mathcal{O}(l^k)$. This compares unfavorably with the rest of the algorithm which is only $\mathcal{O}(l \log l)$ [49:420–422]. Space complexity remains unchanged from simple genetic algorithms. However, the constant term is generally larger and the population size (n) is *much* larger! As is the case with simple genetic algorithms, the time complexity of the evaluation function usually dominates that of the control sequence.

G.5 Fast Messy Genetic Algorithm (fmGA)

The advantage messy GAs have over simple GAs is the ability to create tightly linked building blocks for the optimization of deceptive problems. The disadvantage associated with this better processing is the time complexity of the initialization phase which dominates the mGA algorithm [49:422]. Fast messy GAs are a messy GA variant designed to reduce the complexity of the initialization phase, and thus the overall algorithm time and space complexity [48:59].

G.5.1 Fast Messy Genetic Algorithm Operators. PEI and the selection-only primordial phase of mGAs are replaced by *probabilistically complete initialization* (PCI) and a primordial phase consisting of selection and building block filtering (BBF) in fmGAs. PCI and BBF are an alternate means of providing the juxtapositional phase with highly fit building blocks [48:59–61].

PCI is used to create an initial population whose size is equivalent to the population size at the end of the primordial phase of mGAs. The length of these strings is typically set to $l - k$. The primordial phase then alternately performs several tournament selection

generations to build up copies of highly fit strings followed by BBF to reduce the string length toward the building block size (k). Building block filtering is a simple process that randomly deletes several genes from a string. The juxtapositional phase is the same as in mGAs. A pseudo algorithm for fast messy GAs is shown in Figure G.8.

1. perform probabilistically complete initialization
 evaluate fitness of all population members
2. for $i = 1$ to the maximum number of primordial generations
 perform tournament selection
 if (a building block filtering event is scheduled) then
 perform building block filtering
 evaluate fitness of all population members
 end if
 end loop
3. for $i = 1$ to the maximum number of juxtapositional generations
 perform cut-and-splice
 perform other operators (currently not used)
 evaluate fitness of all population members
 perform tournament selection
 end loop

Figure G.8. Pseudo Algorithm for Fast Messy GAs

G.5.2 Fast Messy Genetic Algorithm Parameters. Fast messy GAs need a building block filtering and thresholding schedule instead of the population size reduction schedule required by mGAs. Goldberg provides formulas for deriving schedules [48], but the formulas contain additional parameters and no guidance is given for choosing their values. The remainder of mGA parameters are used by fmGAs as well.

G.5.3 Mathematical Theory of How (Why) Fast Messy GAs Work. Fast messy GAs are governed by the Schema Theorem (Equation G.2) just like mGAs. The difference relates to how the population of “good” building blocks is created for processing by the juxtapositional phase. Goldberg performs a detailed analysis to show that a much smaller initial population of long strings (PCI) can be manipulated (through BBF) to create a population of “good” building blocks just as effectively as PEI and the primordial phase of mGAs [48].

G.5.3.1 Complexity Analysis. Reducing the overall time complexity of the algorithm is the main reason for switching from mGAs to fmGAs. PCI and BBF result in a time complexity of $\mathcal{O}(l \log l)$ for initialization and the primordial phase combined [48]. Thus, the design goal has been met—fmGAs exhibit better efficiency than mGAs ($\mathcal{O}(l \log l)$ vs $\mathcal{O}(l^k)$) and preserve their effectiveness. Space complexity for fmGAs remains unchanged from SGAs and mGAs ($\mathcal{O}(nl)$) and populations can be sized much smaller than mGAs. Again, the time and space complexities of the evaluation function usually dominate those of the control sequence.

Appendix H. Data Visualization

H.1 Introduction

This appendix delves further into methods previously devised by AFIT students attempting to visualize the landscape for the associated energy minimization of a given protein problem. The bulk of the following information in this appendix is taken directly from Deerman's thesis [25] and his methodology was not employed during this thesis effort.

H.2 Landscape Visualization

Visualizing the search space traversed by the GA in minimizing the energy conformation of the [Met]-Enkephalin, let alone the Polyalanine₁₄ protein, is simply mind-boggling. If we assume the standard AFIT representation of 10 bits per dihedral angle and account for each dihedral angle in [Met]-Enkephalin or Polyalanine on the x-axis, we yield a 240 or 560 bit representation for each chromosome which indicates one energy value on the y-axis. Since the x-axis is discretized, we can reduce this seemingly continuous line into a fixed interval using the natural numbers. Therefore, we have $1.7668e+72$ or $3.7739e+168$ numbers across the bottom of a 2-dimensional graph. The numbers are derived by Equation H.2. This number of x-values makes the problem of visualizing the GA's traversal through the energy landscape beyond the scope of available software tools and computational platforms. On the other hand, the requirement to understand this space remains. The purpose of this section is to explain how we intend to graphically visualize the PSP landscape.

$$Number of Plots = 2^{bit_length} \quad (H.1)$$

where bit_length is the bit representation of the chromosome Number of Plots is total possible enumeration of those bits

Ideally, the best way to visualize the relationship between the molecule and the energy landscape would be in d-dimensional space where d represents the number of dihedral

$$d(f_n, f_m) = \left[\int_{\lambda} \left(|P_{f_n}(D_n, \alpha) - P_{f_m}(D_m, \alpha)|^p + |P_{f_n}(D_n, \alpha) - P_{f_m}(D_m, \alpha)|^p \right) d\alpha \right]^{1/p}$$

Figure H.1. P-norm Equation

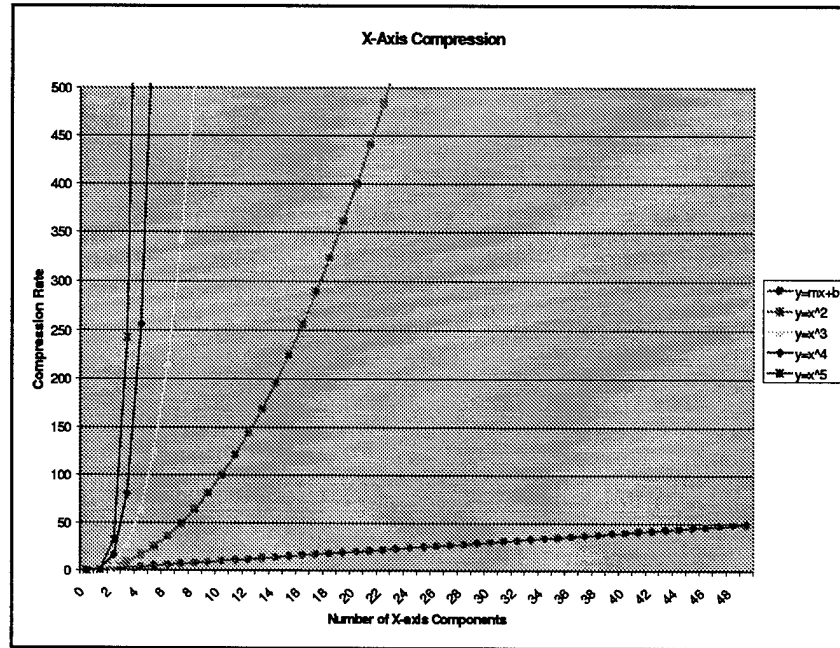


Figure H.2. P-norm Equation

angles. This would reduce the problem to graphing 1024 points on each axis. Alas, there is also no mathematical way to represent 24-dimensional space. Therefore, we have derived a technique to reduce the numerical span of the first purposed visualization methodology to approximately 1934 points which is graphable using MathlabTM.

Mathematically, Deerman used a p-norm projection across the x-axis to make the discrete range of 2240 into a metric data scale using Figure H.1 where $p = 5$. Therefore, each 10-bit representation of a dihedral angle is a separate PFP term within the equation. This transformation ensures that the representational distance between the x-axis values is maintained. Other norms were considered, but since we were trying to compress the data in order to produce a visualization which would easily fit on a single page without a reduction size the 5th-norm worked the best. Figure H.2 shows the compression rate of

Table H.1. Visualization Legend

Color	Range
BLACK (●)	$-\infty \rightarrow +25.0$
BLUE (x)	$+25.01 \rightarrow 1,000$
Yellow (▲)	$1,000.01 \rightarrow 1,000,000$
Red (◆)	$1,000,000.1 \rightarrow \infty$

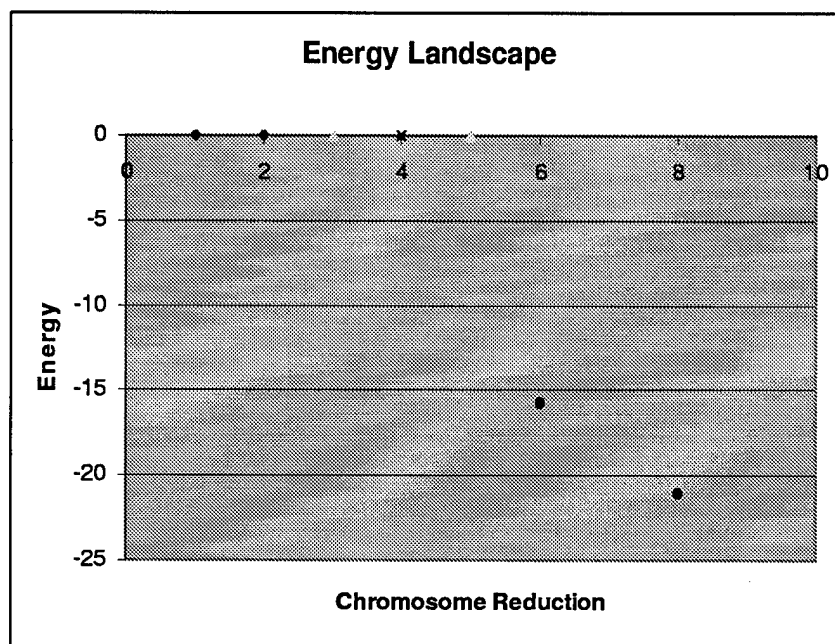


Figure H.3. Visualization Legend

the a few different norms. The quicker the curve grows the more the data is compressed, but the representational distance between any two points on the x-axis is maintained!

The y-axis, on the other hand, still represents the continuous real value range of the energy function. Since we are really interested in a small range of negative values and because it is not uncommon to have a $1e+32$ energy value associated with a molecule, we have chosen to bound the upper limits of the y-axis to +25 kilocalories. The remaining energy values are illustrated upon the graph by exploiting colored graphical gamut's located at zero and the x-axis intersection. Figure H.1 indicates the color meaning and Figure H.3 shows some Deerman's initial test data reduced as stated here.

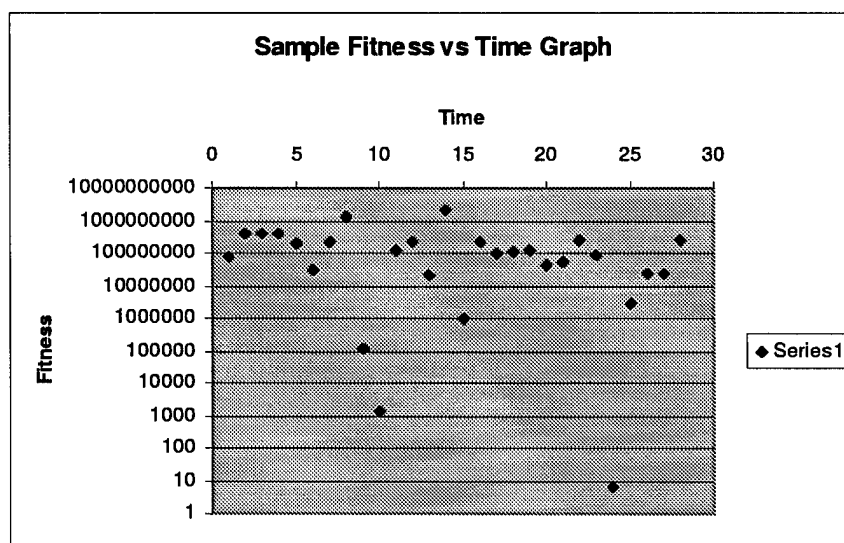


Figure H.4. Sample Fitness vs. Time Graph

A simpler method of mapping out the landscape in a generalized fashion would be to randomly select a number of points from the genotypic domain and calculate their respective fitness values. Plotting the results on a fitness vs. time graph (see Figure H.4) would provide some insight to the overall landscape. The correct mapping of any complex problem's search space is not an easy task and in fact may be an very difficult problem with respect to generating an n -ordered (where " n " is the number of variables) mapping of the landscape as our ability to understand a great number of dimensions is limited; however, any method employed to reduce the dimensionality that is based on sound reasoning and might give a glimpse of what the landscape looks like is beneficial. With this in mind, we have conducted the appropriate tests and focused our visualization efforts, with respect to search space landscape mapping, on obtaining a psuedo-random look at the landscape of both the Met-Enkephalin and Polyalanine proteins through the use of the time vs. fitness mapping of the landscape.

Appendix I. Random Number Generators

I.1 Introduction

The stochastic nature of a GA is totally dependent upon the implemented RNG. Dymek's Appendix A [28] and [110] covers the importance of random number generators due to this heavy reliance. The random number seed dictates where in the problem's search space the GA begins searching and how various segments of the code is implemented. Therefore, it is extremely important that "good" random number generators are used.

I.2 Random Number Generators (RNG)

A good random number generator is defined as one in which no "perfect correlation" occurs [28]; a perfect correlation between two RNGs results in the same instantiated behavior from two separate GA executions. At first glance, this sounds much worse than the situation warrants. For the purposes of validation of experiments, two separate GA test runs, which start with the same random number seed, should result in precisely the same GA behavior. On the other hand, RNGs are only pseudo-random.

The randomness of the sequence of generated "random" numbers depends on the seed. It is possible that two distinct seeds used to initialize a RNG can result in the same or over lapping sequence of random numbers. This results in two separate GA executions having similar behavior, even though different random seeds are used. Not only is it possible but it occurs often, especially if the seeds are potentially related to each other mathematically like (1 and 4096). [110] This is not desired here. Thus, the RNG needs to be check to ensure: 1) that the random numbers produced represent a uniformly distributed set of numbers between the lower and upper bound, 2) that the different seeds used in testing do not correlate, and 3) that within a series of random numbers there is no correlation indicating a relationship between the current random number and a previously generated random number. Uniform distribution guarantees that the random numbers generated have an equal chance of occurrence, as those that are not generated. If two separate random seeds produce correlating sequences of random numbers, two separate executions of the GA using these seeds would search the same problem domain landscape.

Finally, we do not want a correlation within a sequence of random numbers because then our random sequence becomes predictable.

I.2.1 Random Number Generator Correlation. Random number correlation C_l I.1 becomes an issue only when a given algorithm has executed a length of time such that the number of calls to the random number generator function results in the starting over of a given sequence of random numbers. While one would tend to think of this as a sequence of numbers repeating, in reality, the sequence of repeating numbers could be a subset of another set of numbers. For instance, suppose an algorithm makes twenty calls to a random number generator and with the resulting numbers ranged from 0 to 100 are (see Figure I.1. As the figure shows, it is possible to have a subset of those random numbers repeat. This particular phenomenon becomes an issue if the granularity of the application of the random numbers used increases. For instance, one could hardly disagree that if an algorithm randomly select 1's and 0's to create a population member string (chromosome) the odds of having a set of 0's of a given length increase as the length repeatable length decreases in size. If; however, a different algorithm creates a population member based on a much finer definition such as a hexadecimal representation (0-9,A-F) to define the population members. The odds of having a subsequence of those hexadecimal values repeat are less than in the prior case (even as we decrease the repeatable length).

$$C_l = PN_l \quad (I.1)$$

where C_l is the correlation length and PN_l is the Pseudorandom number generator sequence length which is based entirely on the implementation of the PNG.

Another way to look at it would be suppose how many possibilities exist with 2 digits of 1's and 0's. Answer: 4! How many possibilities exist with 2 hexadecimal digits? Answer: 256!. Short of increasing the granularity of the data representation there is very little that can be done with respect to a sub-correlation within a sequence of randomly generated numbers. On the other hand, most algorithms utilize random number generators in a multitude of ways usually with an intermixing of the methods in which they are used.

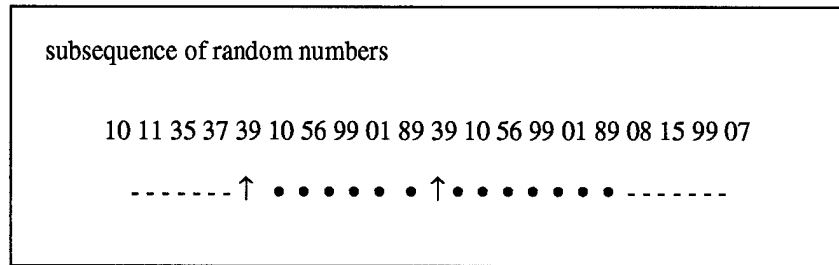


Figure I.1. Repeat of Random Numbers in a Subsequence of Random Numbers

As previously stated correlation is usually thought of with respect to the entire scope of random numbers that can be generated before correlation occurs. For instance suppose a random number generator is base on 32 bit mask. If this is the case, then there are 2^{32} random numbers in the sequence of random numbers before correlation occurs. For most applications this very long correlation factor is sufficient as their specific implementation may not have more than 2^{32} random number generation function calls.

But what about if those algorithms have more than 4 billion calls to a RNG? What is a programmer to do? While there are a number of ways to handle this problem, and this problem does occur more often than expected—especially in a parallel application, there is essentially one way overcome this problem and that is through the use of a number of different RNGs. The idea is to either interweave the calls to the RNGs (either randomly select or in series select a RNG to use) or use the different the RNGs serially (when the first RNG has exhausted all numbers and correlation would occur then a new RNG would be used). [86, 110, 87] Regardless of the method employed the use of different RNGs is a must for those applications that require an inordinate number of random numbers.

Note: In reference [110] the last paragraphs states the following:

"Do not trust the random number generator on your computer unless some random number generator expert guarantees that it is good, and in this case, computer vendors may not count as experts. If possible do your simulation with two different random number generators and compare the results."

With respect to this work, the fmGA algorithm is applying a masking function to generate random numbers and/or the built-in gnu C RNG function call. It is assumed that

their prior work was correct in its implementation and that no correlation with respect to a single run of the fmGA (as the number of RNG calls is sufficiently less than the correlation length) exists.

Appendix J. Additional Design and Implementation Information

J.1 Introduction

The purpose of this appendix is to provide the reader additional insight into the designing of the fmGA. This is done by first looking at the high level design of specific GAs and subsequently the lower design effort. This information is taken verbatim from [41] and he should take credit for the descriptions herein.

J.2 GA High-Level Design

The objective of a good preliminary design is to capture the essence of what needs to be accomplished without regard for specific data structures, control flow, or computer architecture. Several paradigms have been proposed specifically for the specification and design of parallel algorithms, including CSP [61], petri-nets [116], and UNITY [12]. While these methods can be shown to be equivalent, their expressive powers lie in different dimensions. CSP and petri-nets lend themselves more to the specification of control flow interactions. We choose to use UNITY as our design language because it's better suited to specifying data parallelism at high abstraction levels.

UNITY (Unbounded Nondeterministic Iterative Transformations) is a parallel program specification and design language that separates the description of *what* should be done from *when*, *where*, and *how*. A UNITY program (design) describes *what* assignments need to take place, but a mapping to a specific architecture answers the other three questions [12:8-11]. A UNITY program attempts to express the maximum parallelism possible using only assignment statements separated by either parallel bars (\parallel) or a box ($\boxed{\quad}$). (The parallel bars connect assignments that must take place simultaneously. A box separates assignment statements that must be executed at different times.) A time complexity analysis of this maximally parallel design provides a benchmark to evaluate the time complexity realized by various implementations.

Assertions of the form $\{p\} s \{q\}$ are used in UNITY to indicate that the execution of statement s from a state where predicate p is true results in a transition to a state where predicate q is true [12:40]. With the addition of universal and existential quantifiers, and

because of the lack of control flow within UNITY programs, this first order predicate logic is sufficient to prove UNITY programs correct. Then the assertions are *properties* associated with the entire program rather than predicates attached to individual pieces of an algorithm. These properties are classified as either *safety* or *progress* properties that define respectively the legal states and the advancement mechanism(s) of an algorithm.

Program development through UNITY proceeds by translating a specification into a high-level design which is then repeatedly refined until sufficient detail is available for implementation. The proof of correctness for an entire program is built as each refinement is proven correct with respect to its parent design.

We have access to specifications [62, 51, 48] and implementations [56, 88, 96] of simple, messy, and fast messy GAs, however the respective designs are either unavailable or inconsistent with the available software products. This section elaborates high-level UNITY designs for the three GAs, discusses mappings to serial and parallel architectures, and establishes their design time complexities.

With a few minor exceptions, the following notational conventions are observed in these genetic algorithm UNITY designs. Capitalized tokens are either boolean flags or input parameters and can be identified by their context—flags are on the left side of assignment statements and parameters are on the right. Thus, the two can be distinguished from their context. Variables are in *italics* and programming constructs are in regular text.

J.2.1 Simple Genetic Algorithm. Figure J.1 reflects the top-level design of what a simple genetic algorithm should accomplish with the minimum set of constraints on when, where, and how it should be done. Detailed descriptions of what the individual components do are shown in Figures J.2 through J.5 for initialization, selection, crossover, and mutation. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. Noting that all the GA operators are specified using the parallel bars (||) exclusively with no synchronization variables, we can conclude that each operator is also $\mathcal{O}(1)$ given enough processors.

The following properties are evident from the SGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the third

Program SGA

```
declare
  type SGA_string is record {
    genes : array[STRING_LENGTH] of 0..1
    fitness real }
  type Population is list of SGA_string
  old_pop : Population
  new_pop : Population
  gen : integer
initially
  INIT = false
  EVAL = false
  CROSS = true
  MUTATE = true
  SELECT = false
  gen = 0
assign
  <INIT := true /*Generate an initial population*/
    if ( $\neg$ INIT)>
  ||
  <EVAL := true [] old_pop := new_pop /* Evaluate the population*/
    if ( $0 \leq \textit{gen} \leq \textit{MAX\_GENS} \wedge \textit{INIT} \wedge \textit{CROSS} \wedge \textit{MUTATE}$ )>
  ||
  <gen, SELECT, CROSS, MUTATE, EVAL :=
    gen + 1, true, false, false, false /*Select a new population*/
    if ( $0 \leq \textit{gen} \leq \textit{MAX\_GENS} \wedge \textit{EVAL} \wedge \neg \textit{SELECT}$ )>
  ||
  <CROSS := true /*Perform crossover*/
    if ( $0 \leq \textit{gen} \leq \textit{MAX\_GENS} \wedge \textit{SELECT} \wedge \neg \textit{CROSS}$ )>
  ||
  <MUTATE := true /*Perform mutation*/
    if ( $0 \leq \textit{gen} \leq \textit{MAX\_GENS} \wedge \textit{SELECT} \wedge \neg \textit{MUTATE}$ )>
end
```

Figure J.1. UNITY Description of a Simple Genetic Algorithm

Function Initialize

```
assign
  <[|  $i : 0 \leq i < \textit{POP\_SIZE}$  :: /*For each member of the population*/
    <[|  $j : 0 \leq j < \textit{STRING\_LENGTH}$  :: /*Initialize every bit of the string*/
      new_pop[i].genes[j] := Randint(0, 1)) /*to a random value*/
    >
end
```

Figure J.2. UNITY Description of SGA Initialization

Function Select**always**

$$\langle \langle i : 0 \leq i < POP_SIZE ::$$

$$cumulative[i] := \langle +j : 0 \leq j \leq i :: old_pop[j].fitness \rangle \quad /*Relative\ selection\ prob.\ accumulator*/$$
assign

$$\langle \langle i : 0 \leq i < POP_SIZE ::$$

$$new_pop[i] := old_pop[j] \quad /*Roulette\ wheel\ selection*/$$

$$\text{if } (cumulative[j - 1] < \text{Random}(0, cumulative[POP_SIZE - 1]) \leq cumulative[j])$$
end

Figure J.3. UNITY Description of Roulette-Wheel Selection

Function Cross**assign**

$$\langle \langle i : 0 \leq i < POP_SIZE \wedge \text{even}(i) ::$$

$$\langle \exists j : j = \text{Randint}(0, STRING_LENGTH - 2) :: \quad /*Selected\ crossover\ point*/$$

$$new_pop[i].genes, new_pop[i + 1].genes := \quad /*Create\ 2\ children\ from*/$$

$$\quad /*Parent\ \#1\ head\ and\ Parent\ \#2\ tail*/$$

$$\text{concat}(new_pop[i].genes[0..j], new_pop[i + 1].genes[j + 1..STRING_LENGTH - 1]),$$

$$\quad /*Parent\ \#2\ head\ and\ Parent\ \#1\ tail*/$$

$$\text{concat}(new_pop[i + 1].genes[0..j], new_pop[i].genes[j + 1..STRING_LENGTH - 1])$$

$$\text{if } (\text{Random}(0, 1) < CROSSOVER_PROBABILITY)) \rangle \rangle$$
end

Figure J.4. UNITY Description of Single Point Crossover

Function Mutate**assign**

$$\langle \langle i : 0 \leq i < POP_SIZE ::$$

$$\langle \langle j : 0 \leq j < STRING_LENGTH ::$$

$$new_pop[i].genes[j] := (new_pop[i].genes[j] + 1) \bmod 2 \quad /*Change\ a\ bit*/$$

$$\text{if } (\text{Random}(0, 1) < MUTATION_PROBABILITY)) \rangle \rangle$$
end

Figure J.5. UNITY Description of Bitwise Mutation

assignment statement. The fixed point states that *MAX_GENS* generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

invariant

$$0 \leq gen \leq MAX_GENS + 1$$

FP \equiv

$$\langle gen = MAX_GENS + 1 \wedge INIT \wedge SELECT \wedge \neg EVAL \wedge \neg CROSS \wedge \neg MUTATE \rangle$$

progress

$$\neg FP \wedge gen = x \mapsto gen = x + 1$$

All that is required to map the design to a sequential architecture is a specific ordering for all assignment statements (a control structure). At the top level, a sequence similar to the one shown back in Figure G.5 is sufficient. Mapping the lower level designs requires the use of looping control structures and sequences of assignments to implement the quantified and enumerated assignment statements respectively [12:24].

Mappings to parallel architectures require additional design decisions. Most important are the parallel decomposition technique and the granularity of the architecture. However, our high-level design is flexible enough to be mapped to any combination of choices.

A coarse-grained, data-decomposition mapping implements an island model parallel genetic algorithm. For this mapping, regular portions of the assignment statements that are quantified by the population size are allocated to processors so that each processor performs all the assignment operations on a subset of the global population. Internally, each processor contains a serial mapping. Any communication of solutions between processors can be modeled here as a change in the mapping of sets of assignment statements to different processors.

The UNITY design can also be modeled using a master/slave mapping to implement control decomposition. This approach has been considered in cases where the fitness evaluations take significantly longer than the application of the genetic operators. In this configuration, the genetic operator assignment statements might be mapped to the master

processor while the statements that evaluate fitness are distributed to the slaves for computation in parallel. An additional design decision determines whether complete evaluations are assigned to processors or component terms are distributed.

J.2.2 Messy Genetic Algorithm. Figure J.6 reflects the top-level design of what a messy genetic algorithm should accomplish. Detailed descriptions of what the individual components do are shown in Figures J.7 through J.11 for PEI, tournament selection, cut, and splice. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. PEI is also $\mathcal{O}(1)$ given enough processors due to the exclusive use of parallel bars (\parallel). Tournament selection and cut-and-splice however, are $\mathcal{O}(POP_SIZE)$ because the specification calls for selection without replacement and constant population size during cut-and-splice. Given that population size has been shown to be proportional to string length [47], we conclude that these two operators are $\mathcal{O}(l)$.

The following properties, similar to the simple GA's, are evident from the mGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the second assignment statement. The fixed point states that *MAX_GENS* generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

invariant

$$0 \leq gen \leq MAX_GENS + 1$$

FP \equiv

$$\langle gen = MAX_GENS + 1 \wedge INIT \wedge TOURNAMENT \wedge \neg CUT_AND_SPLICE \rangle$$

progress

$$\neg FP \wedge gen = x \mapsto gen = x + 1$$

Mappings to serial and parallel architectures based on decomposition techniques and granularity are similar to SGA mappings. A serial mapping would result in an algorithm that resembles the one shown before in Figure G.7. It's also possible to map PEI, the primordial phase, and the juxtapositional phase independently so that each would be

Program MGA

declare

type *MGA_string* is record {
alleles : array[*STRING_LENGTH* · *EXTENSION*] of integer range *CARDINALITY*
loci : array[*STRING_LENGTH* · *EXTENSION*] of integer range *STRING_LENGTH*
fitness : real }
type *Population* is list of *MGA_string*
old_pop : *Population*
new_pop : *Population*

always

C = *CARDINALITY*
k = *BLOCK_SIZE*
l = *STRING_LENGTH*
 $POP_SIZE = C^k \binom{l}{k}$

initially

INIT = false
CUT_AND_SPLICE = false
TOURNAMENT = false
gen = 0
curr_pop_size = *POP_SIZE*

assign

⟨*INIT* := true /*Generate all building blocks—PEI*/
if (¬*INIT*)⟩
||
⟨*TOURNAMENT*, *gen* := (*gen* = *PRIMORDIAL_GENS*), *gen* + 1 /*Primordial*/
if (0 ≤ *gen* ≤ *PRIMORDIAL_GENS* ∧ *INIT* ∧ ¬*TOURNAMENT*)⟩ /*Phase*/
||
⟨*CUT_AND_SPLICE*, *TOURNAMENT* := true, false
if (*PRIMORDIAL_GENS* < *gen* ≤ *MAX_GENS* ∧ /*Juxtapositional*/
TOURNAMENT ∧ ¬*CUT_AND_SPLICE*)⟩
||
⟨*TOURNAMENT*, *CUT_AND_SPLICE*, *gen* := true, false, *gen* + 1
if (*PRIMORDIAL_GENS* < *gen* ≤ *MAX_GENS* ∧ /*Phase*/
CUT_AND_SPLICE ∧ ¬*TOURNAMENT*)⟩

end

Figure J.6. UNITY Description of a Messy Genetic Algorithm

Function PEI**declare**type *BB* is array[*k*] of integer range *C**building_blocks* : array[C^k] of *BB***always** $\langle \langle i : 0 \leq i < C^k ::$ $\langle \langle j : 0 \leq j < k ::$ *building_blocks*[*i*][*j*] = *i* mod *j* $\rangle \rangle$ /*All C^k values for *k*-size building blocks*/*combinations* = list *x* | *x* $\in \mathcal{P}(\{0, 1, \dots, l-1\}) \wedge |x| = k$ /**l* choose *k* combinations*/**assign** $\langle \langle i : 0 \leq i < \binom{l}{k} ::$ $\langle \langle j : 0 \leq j < C^k ::$ *new_pop*[(*i* · C^k) + *j*].*loci*, *new_pop*[(*i* · C^k) + *j*].*alleles* :=*combinations*[*i*], *building_blocks*[*j*] $\rangle \rangle$ /*Create initial population*/

□

EV AL := true □ *old_pop* := *new_pop* \rangle /*Evaluate initial population*/**end**

Figure J.7. UNITY Description of Partially Enumerative Initialization

implemented differently. This piecewise mapping creates more design choices for data distribution and communication strategies for mGAs.

J.2.3 Fast Messy Genetic Algorithm. Figure J.12 reflects the top-level design of what a fast messy genetic algorithm should accomplish. Detailed descriptions of what PCI and tournament selection do are shown in Figures J.13 and J.14. Cut-and-splice remains the same as in a messy GA. The design for *evaluation* has been omitted because it is problem dependent. Assuming a fixed number of generations, the high-level control is $\mathcal{O}(1)$. PCI and the primordial phase are now $\mathcal{O}(l)$ because the building blocks can be generated from a much smaller initial population using tournament selection and building block filtering. Tournament selection and cut-and-splice are still $\mathcal{O}(\text{POP_SIZE})$ for the same reasons as their messy GA counterparts. After applying the $\text{POP_SIZE} \propto l$ transformation, all operators are $\mathcal{O}(l)$.

The following properties, similar to the mGA's, are evident from the fmGA specification. The invariant is derived from the initial state for *gen* and monotonically incrementing *gen* in the second assignment statement. The fixed point states that *MAX_GENS* generations will be completed then processing will stop. To show progress we note that if the program is not at a fixed point, then *gen* must increase.

Function *Tournament_Selection*

```

always
   $\lambda_i$           =  $|old\_pop[i].alleles|$ 
   $\theta_{ij}$          =  $\left\lceil \frac{\lambda_i \lambda_j}{i} \right\rceil$ 
   $compatible_{ij}$  =  $|cand_i.loci \cap cand_j.loci| \geq \theta_{ij}$ 
   $comp_i$         =  $(\langle \min j : i < j \leq i + n_{sh} \wedge i < j < curr\_pop\_size \wedge compatible_{ij} = true :: j \rangle)$ 
initially
   $curr$           = 0
   $new\_index$      = 0
   $sequencer$      = 0
   $n_{sh}$           = SHUFFLES
assign
   $\langle curr\_pop\_size := \frac{curr\_pop\_size}{REDUCTION\_FACTOR} \quad /*Reduce\ the\ population\ size*/$ 
     $\text{if } (0 < gen \leq PRIMORDIAL\_GENS \wedge gen \bmod REDUCTION\_RATE = 0) \rangle$ 
   $\langle$ 
     $\langle \langle i : 0 \leq i < curr\_pop\_size ::$ 
       $\langle \exists j : j = Randint(i, curr\_pop\_size - 1) ::$ 
         $old\_pop[i], old\_pop[j], sequencer := old\_pop[j], old\_pop[i], 1$ 
         $\text{if } (sequencer = 0) \rangle \quad /*Permute\ the\ population\ order*/$ 
       $\rangle$ 
     $\rangle$ 
     $\langle$ 
       $\langle new\_pop[new\_index], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index, curr :=$ 
         $old\_pop[curr], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$ 
         $/*Choose\ a\ string\ into\ the\ next\ generation\ based\ on\ its\ fitness\ and\ length*/$ 
         $\text{if } (comp_{curr} \leq curr + n_{sh} \wedge$ 
           $(old\_pop[curr].fitness \text{ betterthan } old\_pop[comp_{curr}].fitness \vee$ 
           $(old\_pop[curr].fitness = old\_pop[comp_{curr}].fitness \wedge \lambda_{curr} < \lambda_{comp_{curr}}))) \sim$ 
           $old\_pop[comp_{curr}], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$ 
           $\text{if } (comp_{curr} \leq curr + n_{sh} \wedge$ 
             $(old\_pop[comp_{curr}].fitness \text{ betterthan } old\_pop[curr].fitness \vee$ 
             $(old\_pop[comp_{curr}].fitness = old\_pop[curr].fitness \wedge \lambda_{comp_{curr}} < \lambda_{curr}))) \sim$ 
             $old\_pop[curr], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index + 1, curr + 1$ 
             $\text{if } (comp_{curr} > curr + n_{sh}) \rangle$ 
           $\text{if } (new\_index < curr\_pop\_size \wedge sequencer = 1) \rangle$ 
         $\rangle$ 
       $\langle sequencer := 0 \quad /*Permute\ population\ again*/$ 
         $\text{if } (new\_index < curr\_pop\_size \wedge curr \geq curr\_pop\_size) \rangle$ 
       $\rangle$ 
       $\langle old\_pop := new\_pop$ 
         $\text{if } (new\_index = curr\_pop\_size) \rangle$ 
     $\rangle$ 
   $\rangle$ 
end

```

Figure J.8. UNITY Description of mGA Tournament Selection

Function *Cut_and_Splice***declare***cut_list* : list of *MGA_STRING***initially***new_index* = 0*sequencer* = 0*curr* = 0*CUT* = false*SPLICE* = true**assign** $\langle \forall i : 0 \leq i < \text{curr_pop_size} ::$ $\langle \exists j : j = \text{Randint}(i, \text{curr_pop_size} - 1) ::$ *old_pop*[*i*], *old_pop*[*j*], *sequencer* := *old_pop*[*j*], *old_pop*[*i*], 1if (*sequencer* = 0))) /*Permute the population order*/

||

 $\langle \text{CUT}, \text{SPLICE}, \text{curr}, \text{sequencer} :=$ true, false, *curr*, *sequencer* /*Perform a cut operation*/if ($\neg \text{CUT} \wedge \text{SPLICE} \wedge \text{curr} < \text{POP_SIZE} \wedge \text{sequencer} = 1$) ~false, true, *curr* + 2, *sequencer* /*Perform a splice operation*/if ($\text{CUT} \wedge \neg \text{SPLICE} \wedge \text{curr} < \text{POP_SIZE} \wedge \text{sequencer} = 1$) ~*CUT*, *SPLICE*, 0, 0 /*Permute population again*/if (*curr* ≥ *POP_SIZE*)**end**

Figure J.9. UNITY Description of Cut and Splice

Function *Splice***initially***cut_index* = head**assign** $\langle \text{new_population}[\text{new_index}].\text{allele}, \text{new_population}[\text{new_index}].\text{loci}, \text{cut_index} :=$

/*Copy single string if only one string left or splice probability not met*/

cut_list[*cut_index*].*allele*, *cut_list*[*cut_index*].*loci*, *cut_index* + 1if (*cut_index* = tail - 1 ∨ Random(0,1) > *P_s*) ~

/*Splice two strings together otherwise*/

concat(*cut_list*[*cut_index*].*allele*, *cut_list*[*cut_index* + 1].*allele*),concat(*cut_list*[*cut_index*].*loci*, *cut_list*[*cut_index* + 1].*loci*), *cut_index* + 2

otherwise)

end

Figure J.10. UNITY Description of Splice

Function Cut**always** $\lambda_i = |\text{old_pop}[i].\text{alleles}|$ **assign**

```

( (∃j : j = Randint(0, λi - 2) :: /*Cut first mate & save pieces, based on cut probability*/
  cut_list[head].loci, cut_list[head].alleles, cut_list[tail].loci, cut_list[tail].alleles :=
    old_pop[i].loci[0..j], old_pop[i].alleles[0..j],
    old_pop[i].loci[j + 1..λi - 1], old_pop[i].alleles[j + 1..λi - 1]
    if (Random(0,1) < Pcλi) ~
      old_pop[i].loci, old_pop[i].alleles, null, null
    otherwise)

```

||

```

(∃j : j = Randint(0, λi+1 - 2) :: /*Cut second mate & save pieces, based on cut probability*/
  cut_list[tail - 1].loci, cut_list[tail - 1].alleles, cut_list[head + 1].loci, cut_list[head + 1].alleles :=
    old_pop[i + 1].loci[0..j], old_pop[i + 1].alleles[0..j],
    old_pop[i + 1].loci[j + 1..λi+1 - 1], old_pop[i + 1].alleles[j + 1..λi+1 - 1]
    if (Random(0,1) < Pcλi+1) ~
      old_pop[i + 1].loci, old_pop[i + 1].alleles, null, null
    otherwise))

```

□

```

(total_strings := /*Tally the final number of string segments*/
  4 if (cut_list[head + 1].loci = cut_list[tail].loci = null) ~
  2 if (cut_list[head + 1].loci ≠ null ∧ cut_array[tail].loci ≠ null) ~
  3 otherwise)

```

end

Figure J.11. UNITY Description of Cut

Program FMGA

declare

```

    type MGA_string is record {
        alleles : array[STRING_LENGTH · EXTENSION] of integer range CARDINALITY
        loci    : array[STRING_LENGTH · EXTENSION] of integer range STRING_LENGTH
        fitness : real }
    type BBF_schedule is record {
        gen  : integer
        λ    : integer
        θ    : integer }
    type Population is list of MGA_string
    old_pop : Population
    new_pop : Population

```

always

```

    C      = CARDINALITY
    k      = BLOCK_SIZE
    l      = STRING_LENGTH
    na    = Na
    ng    =  $\left(\frac{l}{l-k}\right)^k$ 
    pop_size = na · ng

```

initially

```

    INIT           = false
    CUT_AND_SPLICE = false
    TOURNAMENT     = false
    gen            = 0

```

assign

```

    ⟨INIT := true   if (¬INIT)⟩    /*Generate initial population*/
    ||
    ⟨TOURNAMENT, gen := (gen = PRIMORDIAL_GENS), gen + 1    /*Primordial*/
      if (0 ≤ gen ≤ PRIMORDIAL_GENS ∧ INIT ∧ ¬TOURNAMENT)⟩  /*Phase*/
    ||
    ⟨CUT_AND_SPLICE, TOURNAMENT := true, false
      if (PRIMORDIAL_GENS < gen ≤ MAX_GENS ∧
          TOURNAMENT ∧ ¬CUT_AND_SPLICE)⟩                    /*Juxtapositional*/
    ||
    ⟨TOURNAMENT, CUT_AND_SPLICE, gen := true, false, gen + 1
      if (PRIMORDIAL_GENS < gen ≤ MAX_GENS ∧
          CUT_AND_SPLICE ∧ ≠ TOURNAMENT)⟩                  /*Phase*/

```

end

Figure J.12. UNITY Description of a Fast Messy Genetic Algorithm

Function PCI**declare***loci_array* : array[*l*] of integer**initially** $\langle \parallel i : 0 \leq i < l ::$ *loci_array*[*i*] = *i* \rangle $\langle \parallel i : 0 \leq i < POP_SIZE ::$ *GENERATED*_{*i*} = false \rangle **assign** $\langle \parallel i : 0 \leq i < pop_size ::$ $\langle \parallel j : 0 \leq j < l - k ::$ $\langle \exists x : x = Randint(j, l - 1) ::$ /*Randomly generate loci and alleles*/*new_pop*[*i*].*loci*[*j*], *new_pop*[*i*].*alleles*[*j*], *loci_array*[*j*], *loci_array*[*x*], *GENERATED*_{*i*} :=*loci_array*[*x*], Randint(0, *C* - 1), *loci_array*[*x*], *loci_array*[*j*], trueif (\neg *GENERATED*_{*i*}) $\rangle \rangle$ \parallel $\langle EVAL := true \parallel old_pop := new_pop$ /*Evaluate the new population*/if ($\langle \parallel i : 0 \leq i < POP_SIZE :: GENERATED_i \rangle$) \rangle **end**

Figure J.13. UNITY Description of Probabilistically Complete Initialization

invariant $0 \leq gen \leq MAX_GENS + 1$ **FP** \equiv $\langle gen = MAX_GENS + 1 \wedge INIT \wedge TOURNAMENT \wedge \neg CUT_AND_SPLICE \rangle$ **progress** $\neg FP \wedge gen = x \mapsto gen = x + 1$

Mappings to serial and parallel architectures based on decomposition technique and granularity are similar to mGA mappings. A serial mapping would result in an algorithm that resembles the one shown in Figure G.8. PCI, the primordial phase, and the juxtapositional phase can again be independently mapped so that each would be implemented differently. The same options for data distribution and communication strategies apply from mGAs.

J.3 GA Low-Level Design and Implementation

Specific data and control structures are the items to be addressed during low-level design and implementation. The following sections describe these major components of each genetic algorithm. In each case, the flexibility of the target language ('C') has been used to full advantage. This is especially evident in the initial use of pointers and dynamic

Function *Tournament_Selection*

```

always
   $\lambda_i$           =  $|old\_pop[i].alleles|$ 
   $\theta_{ij}$          =  $\left\lceil \frac{\lambda_i \lambda_j}{l} \right\rceil$ 
   $compatible_{ij}$  =  $|cand_i.loci \cap cand_j.loci| \geq \theta_{ij}$ 
   $comp_i$         =  $(\langle \min j : i < j \leq i + n_{sh} \wedge i < j < pop\_size \wedge compatible_{ij} = true :: j \rangle)$ 
initially
   $curr$           = 0
   $new\_index$      = 0
   $sequencer$      = 0
   $n_{sh}$           = SHUFFLES
   $next\_filter$    = 1
assign
  (  $\langle \langle i : 0 \leq i < pop\_size ::$ 
    (  $\langle j : 0 \leq j < schedule[next\_filter].\lambda - schedule[next\_filter - 1].\lambda ::$ 
      (  $\langle \exists x : x = Randint(0, \lambda_i) ::$ 
        (  $\langle \langle y : x < y < \lambda_i ::$  /*Perform BBF*/
          (  $old\_pop[i].loci[y - 1], old\_pop[i].alleles[y - 1] :=$ 
            (  $old\_pop[i].loci[y], old\_pop[i].alleles[y] \rangle \rangle \rangle$ 
          if ( $gen = schedule[next\_filter].gen$ )
        )
      )
    )
  )
  (  $\langle i : 0 \leq i < pop\_size ::$ 
    (  $\langle j : j = Randint(i, pop\_size - 1) ::$  /*Permute the population order*/
      (  $old\_pop[i], old\_pop[j], sequencer := old\_pop[j], old\_pop[i], 1$ 
        if ( $sequencer = 0$ )
      )
    )
  )
  (  $\langle new\_pop[new\_index], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index, curr :=$ 
    (  $old\_pop[curr], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$ 
      /*Choose a string into the next generation based on its fitness and length*/
      if ( $comp_{curr} \leq curr + n_{sh} \wedge$ 
        (  $old\_pop[curr].fitness$  betterthan  $old\_pop[comp_{curr}].fitness \vee$ 
          (  $old\_pop[curr].fitness = old\_pop[comp_{curr}].fitness \wedge \lambda_{curr} < \lambda_{comp_{curr}}$  ) ) ) ~
        (  $old\_pop[comp_{curr}], old\_pop[comp_{curr}], old\_pop[curr + 1], new\_index + 1, curr + 2$ 
          if ( $comp_i \leq curr + n_{sh} \wedge$ 
            (  $old\_pop[comp_{curr}].fitness$  betterthan  $old\_pop[curr].fitness \vee$ 
              (  $old\_pop[comp_{curr}].fitness = old\_pop[curr].fitness \wedge \lambda_{comp_{curr}} < \lambda_{curr}$  ) ) ) ~
            (  $old\_pop[curr], old\_pop[curr + 1], old\_pop[comp_{curr}], new\_index + 1, curr + 1$ 
              if ( $comp_{curr} > curr + n_{sh}$ )
            )
          if ( $new\_index < pop\_size \wedge sequencer = 1$ )
        )
      )
    )
  )
  (  $sequencer := 0$  if ( $new\_index < pop\_size \wedge curr \geq pop\_size$ ) ) /*Permute population again*/
  (  $old\_pop := new\_pop$  if ( $new\_index = pop\_size$ ) )
end

```

Figure J.14. UNITY Description of fmGA Tournament Selection

memory allocation to accommodate run-time specification of data structure sizes, but subsequent access of those data structures using array notation.

J.3.1 Simple Genetic Algorithm. The control structure used by Genesis is a direct instantiation of the algorithm presented in Figure G.5. The high-level design (Figure J.1) is mapped to a sequence of statement enclosed inside a loop from 0 to the maximum number of generations. The string data structure contains both the genes and a location to store the fitness.

```
typedef struct {  
    char *Gene;  
    double Perf;  
    int Needs_evaluation;  
} STRUCTURE;
```

Two populations are dynamically allocated during initialization, one to hold the current population and one accept the new strings created by the recombination operators. The dual access modes that 'C' provides enable the programmer to optimize specific operations. Direct access to individuals in the population is available through the array-indexing mode. A newly created population can be moved to the old population by swapping the pointers to each population. One of these two operations would be significantly slower if only one access mode were available.

J.3.2 Messy and Fast Messy Genetic Algorithms. The control structures for the mGA and fmGA are direct instantiations of their respective algorithms presented in Figures G.7 and G.8. Their high-level designs (Figures J.6 and J.12) are mapped in the same fashion as the simple GA. The string data structure is modified to store the additional locus information.

```
typedef struct {  
    int    *locus;  
    char   *allele;
```

```
    double fitness;  
} mgastring;
```

The population data structures are implemented exactly like the simple GA population with all the associated benefits.

J.4 Genetic Algorithm Fitness Functions for Energy Minimization

A major objective in any research effort is to be able to compare results to previous research. In order to make these comparisons, the current effort must provide accurate results and the relationships between compared works must be understood. This section discusses the accuracy of our energy model/fitness function. Appendix ?? contains details on the relationship between data files and procedures for comparing results obtained from the two different energy models used in previous research [106, 4].

The design and implementation of a fitness function is critical to the successful application of genetic algorithms to a problem domain. For most complex problems of interest, the majority of the total execution time is spent evaluating the quality of solutions. Thus it is imperative that the detailed design and implementation are as efficient as possible. In the research environment however, some inefficiency may be tolerated to take advantage of other software characteristics. In particular, we prefer to remain isolated from machine dependent math libraries in the interest of enhancing portability.

J.4.1 Previous Energy Model Designs. Two energy model implementations are available for our use with genetic algorithms: ECEPP and CHARMM. A description of the ECEPP design and implementation is given in [29]. There are several drawbacks to using ECEPP as a general energy model. The greatest hindrance is the limited size (≈ 50 amino acids) and types of proteins that it can accommodate [29]. The configuration management complexity of this implementation is also very high because we have to interface C and Fortran code, and portability is poor because ECEPP requires special math libraries that aren't widely available on all AFIT computer systems.

The design specification for CHARMM is given in Brooks [7], however the only implementation is proprietary (QUANTA [13]) so AFIT designed and implemented its own

Table J.1. Energy Component Comparison

Implementation Component	Previous Implementation (kcal/mol)	QUANTA Values (kcal/mol)
Bond (ρ)	12.380	12.374
Bond Angle (θ)	6.189	6.183
Dihedral Angle (ϕ)	202.479	8.204
Lennard-Jones	-45.133	-15.014
Electrostatic	-0.267	-40.973
Total	175.649	-29.225

software for use with genetic algorithms [4]. Because it's the only force field energy model capable of modeling general organic molecules, the CHARMM energy model is essential to the Air Force's research of non-linear optical (NLO) materials. The design specifies list and array data structures that are suitably efficient for the highly sequential, read-only access required by the majority of the program. Our design is: scalable over all protein sizes (within machine limits) and types, written in C for simple integration with our GAs, and requires nothing more than standard C header files.

ECEPP and CHARMM are both force-field energy models so we expect their time complexity to be $\mathcal{O}(n^2)$, where n is the number of atoms in the protein, because force-field models account for all pairwise interactions between atoms [81, 82]. Our CHARMM implementation uses lists of the pairwise interactions and constant time calculations of the energy components to realize the minimum time complexity of the design. The ECEPP implementation is also $\mathcal{O}(n^2)$ [29].

J.4.2 Energy Model Design Enhancements. The model created by [4] was roughly 90% accurate as an implementation of the CHARMM energy minimization function and data structures. Table J.1 compares the individual energy components from this model with the values obtained from QUANTA for a specific conformation of [Met]-enkephalin. The design enhancements discussed in the following paragraphs have been implemented to eliminate the discrepancies between the two implementations.

Since the bond lengths and bond angles are being held fixed, we'd expect no differences between the two CHARMM implementations for those energy terms. The differences

shown are less than one tenth of one percent and are at least three orders of magnitude smaller than the relative error of the other terms. The dihedral angle energy is the greatest source of error, followed by the Lennard-Jones potential and electrostatic terms. A technical review of this CHARMM energy model design and implementation uncovered several deficiencies that have been resolved as part of this effort.

1. There are actually three classes of dihedral angles—fixed, independent, and *dependent*. Dependent dihedral angles weren't handled in the original design. Any rotation of an independent dihedral angle without a corresponding rotation of its dependent dihedral angles (Figure J.15) results in a higher energy value for the system. Figure J.16 illustrates how the molecules should react to a rotation of the independent dihedral angle.
2. QUANTA treats the two atoms at the ends of a group defining a dihedral angle as a special case, non-bonded interaction. The energy contribution of this *1-4 non-bonded interaction* is calculated as one-half of the normal non-bonded energy value between non-bonded atom pairs.
3. Dihedral angles were encoded assuming a range of $-\pi$ to π , however they were then mistakenly decoded in the range of 0 to 2π .
4. The dihedral energy term should include a parameter for the *periodicity* of the angle. Together, the periodicity (n) and γ (see Equation ??) define the number of minimum energy dihedral angles and their values (see Figure J.17). Only periodicities of $n = 1, 2, 3, 4$, and 6 are allowed, although a single dihedral angle might have more than one periodicity associated with it.
5. A units conversion factor of ≈ 332.0 was missing from the electrostatic energy calculation. This factor represents a relative dielectric constant of one ($\epsilon_{rel} = 1$).

J.4.3 Implementation Details. Our C implementation of the CHARMM energy model reflects the previously noted priority of portability over efficiency during this research phase. While highly optimized matrix-matrix and matrix-vector operations are available in various math libraries [66, 77], they weren't incorporated. Instead C versions were written to enhance machine portability. This has paid dividends in the ease with which both the GA and energy function code has been ported from SPARCstations to Silicon Graphics and IBM PCs as well as the iPSC2, iPSC860, and Paragon parallel computing platforms.

Certain minor inefficiencies have been introduced by the modifications required to validate the energy model. These inefficiencies are either one-time costs incurred during

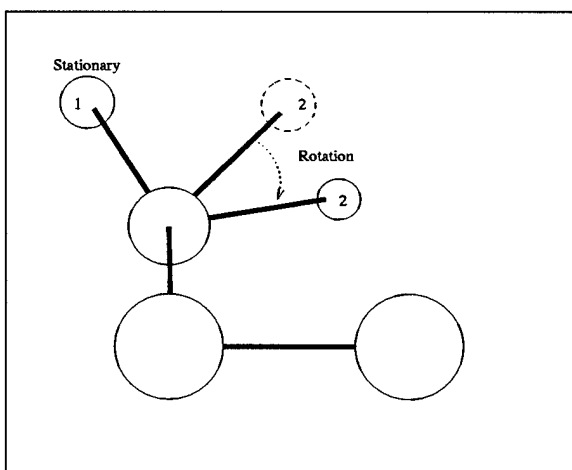


Figure J.15. Incorrect Dependent Dihedral Angle Rotation: Dependent atom (1) remains stationary while independent atom (2) rotates.

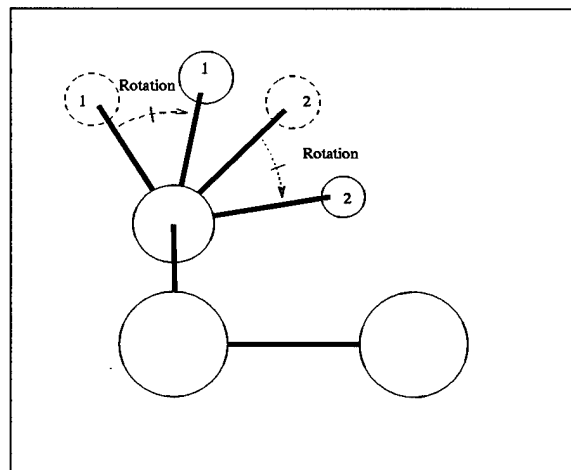


Figure J.16. Correct Dependent Dihedral Angle Rotation: Dependent atom (1) rotates in unison with the rotation of independent atom (2).

data structure initialization or a small number of recurring costs during the energy calculation. In either case, the code is documented to identify the source of the inefficiency and any known possible solutions to the problem. No modifications have resulted in an increase in the overall time complexity of the energy function which has been shown consistent with the $\mathcal{O}(n^2)$ prediction from the design [40].

Item 1 on the deficiencies list is the most difficult modification to implement. It requires the addition of two fields to the *ATOM_TYPE* data structure and additional initialization code. Each dihedral angle must identify an independent dihedral angle that it's dependent on and the angular value of that relationship. Correcting this item won't change any of the energy terms shown in Table J.1, however, this change is critical to correctly calculating the coordinates of atoms in the protein due to rotations of the independent dihedral angles during GA processing. Without this change, the genetic algorithm would attempt to minimize the difference between population members and the input rather than optimize the conformation!

Item 2 requires an additional pairwise interaction list to process the 1-4 non-bonded interactions. The creation of this list can be easily inserted into the section that generates

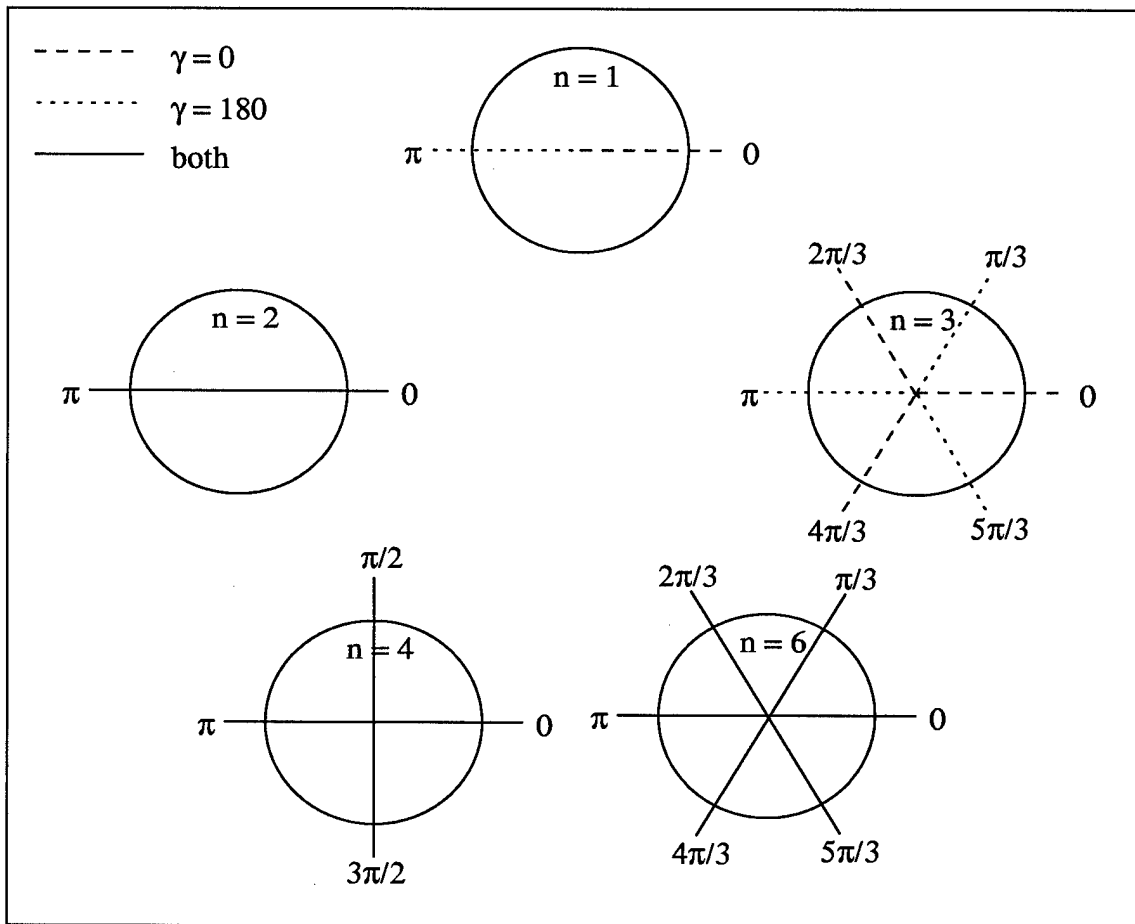


Figure J.17. Dihedral Angle Periodicity

the other interaction lists and processing the list requires only one additional call to the function that calculates non-bonded interaction energies. The only difference between these non-bonded interactions and all the others on the *NON_BONDED* list is the fact that the results for 1-4 non-bonded interactions are scaled by one half. This modification will change the values returned for the electrostatic and Lennard-Jones energy terms.

Items 3, 4 and 5 from the deficiencies list are simple, one-line changes represented by transformations J.1 through J.3.

$$dih \cdot (1 + \cos(dih_ang - \gamma)) \Rightarrow dihed \cdot (1 + \cos((n \cdot dih_ang) - \gamma)) \quad (J.1)$$

$$atom[i].dih_ang = \left(\frac{(360\pi)temp}{180 \cdot 2^{slice}} \right) \Rightarrow atom[i].dih_ang = \left(\frac{(2\pi)temp}{2^{slice}} \right) - \pi \quad (J.2)$$

$$\frac{(atom[i].chg)(atom[j].chg)}{r} \Rightarrow \frac{(atom[i].chg)(atom[j].chg)prop_const}{r} \quad (J.3)$$

The first two changes should correct the dihedral energy term. If the last modification were made in isolation the electrostatic energy term would still be incorrect ($-0.267 \times 332 = -88.644 \not\approx -40.973$). This change combined with the resolution of deficiency 2 should correct both the electrostatic and Lennard-Jones potential.

The energy values obtained from the original energy model didn't exhibit any correlation with the values obtained from QUANTA for identical protein structures. A strong correlation between the two implementations is the goal we seek to obtain with these energy model enhancements.

J.5 Summary

Software reuse is a major design decision for genetic algorithm implementations at AFIT. Effort is expended on design recapture using UNITY to aid in the understanding of that software. The designs presented in this chapter have been created using a top-down design methodology based on the existing 'C' code. The performance of computer programs is judged by their efficiency and effectiveness. This chapter also developed and compared the calculated efficiencies of implementations with the design-predicted efficiencies for the

major components of three genetic algorithms. The effectiveness of the CHARMM energy implementation is examined and a plan to improve its accuracy is designed. The next chapter describes the test setup to measure these performance characteristics.

J.6 AFIT Implementation of the fmGA PSP Software

AFIT has more than 10 years of research effort with respect to finding a viable solution to the geometrical conformation of the Met-Enkephalin protein. Previous master student's efforts have played an important part in our "collective" understanding of the Protein Structure Prediction problem, as well as our ability to employ genetic algorithms to finding a viable solution.

The present AFIT implementation (see Figure ??) is coded in C and is located in the genetic directory in the Parallel Lab's account (Room 243, Bldg. 640, Wright Patterson AFB).

There are a number of divisions of the Toolkit (see Figure 3.1) - Visualization, Binary Genetic Algorithms, Fast Messy Genetic Algorithms, Multi-Chromosome Algorithms, Real-Value Genetic Algorithms, Linkage Friendly Genetic Algorithms, Messy Genetic Algorithms, Permutation Genetic Algorithms, and Multi-Objective Genetic Algorithms Simple.

J.6.1 Inputs to AFIT Toolkit. There are four distinct input files used to handle the necessary problem domain data relative to the PFP and the AFIT Toolkit-three of which are generated by two software packages (see figure below). Cerius2 produces a sequential listing of all atoms present in the molecule, which for all practical purposes is known as the Z-matrix file and is formatted, line-by-line, in accordance with the structure outlined in figure 5. The bond length is the distance between the present atom and atomj. The bond angle is formed between the present atom, atomj, and atomk. Figure 18 - Input/Output of AFIT Toolkit

The dihedral is the torsion angle of the middle bond formed between the present atom, atom_j, atom-*k*, and atom_i. Figure 6 provides an example and shows that, for this specific line, we are dealing with a Carbon atom whose distance is 1.4994 angstroms (an angstrom equals 10⁻¹⁰ meters) meters from the previous atom in the list-in this case a

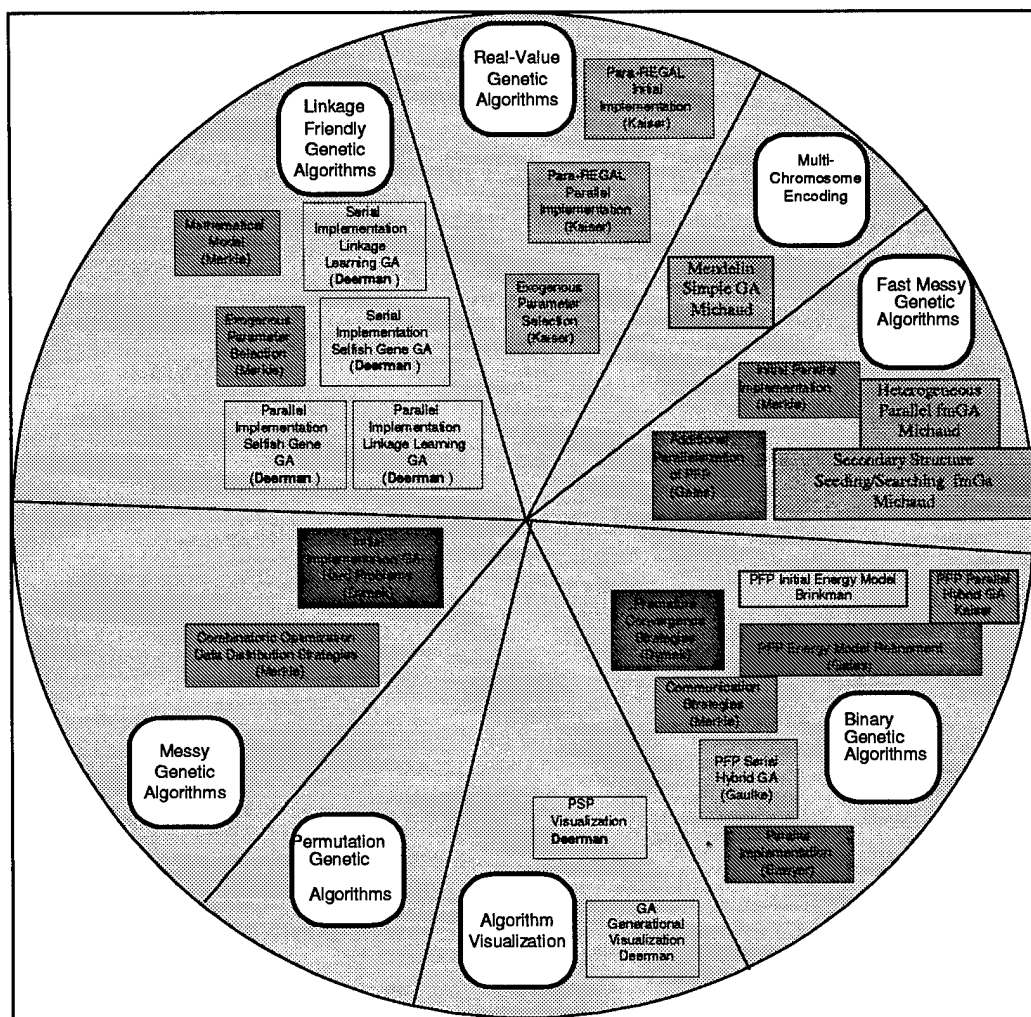


Figure J.18. AFIT's ToolKit

[atom type][bond length][flag][bond angle][flag][dihedral][flag][atomj][atoml][charge]

Figure J.19. Z-matrix File Format

```

N 1.35599 0 121.56565 0 -117.09099 1 3 2 1 0.000
C 1.49994 0 111.60606 0 -119.97103 1 3 2 1 0.000

```

Figure J.20. Example lines of Z-matrix File

Nitrogen atom. There is a 111.60606 degree angle-with vertex at atom 3-this is formed between the present atom, atom 3, and atom 2. Also, there is a -119.97103-degree dihedral angle formed on the bond between atom 3 and atom 2 with respect to the chain of atoms extending from the present atom, the carbon atom in this case, to atom 1. The flags are used to indicate whether the parameters are held constant "0" or variable "1". The charge field of the Z-matrix input file is not used. The residue topology file (RTF), is produced by a package called QUANTA and is used to supply the AFIT Toolkit information specifically addressing atomic charges and atom-type information.

The third file, also derived from the QUANTA software package, is the parameter (PARM) file and is used to provide constant parameters associated with bond lengths, bond angles, dihedral angles, and non-bonded pairs [43].

J.6.2 Outputs from AFIT Toolkit. There are three output files that the AFIT Toolkit generates; out, PDB, and user-generated output which usually goes to the screen; however, it can be redirected to a user-specified file. The out file contains a line for every generation containing data such as the number of trials, percent converged, minimum energy at that generation, and the average energy of that generation. The PDB file contains Cartesian coordinates corresponding to each atom. The Cartesian coordinates are important for the evaluation of the energy function of a particular conformation. Finally, the user-generated output is derived by the inclusion of printf statements in the code [43].

Bibliography

1. Bäck, Thomas and others. "Evolutionary Programming and Evolution Strategies: Similarities and Differences." *The Second Annual Conference on Evolutionary Programming*. 11-22. San Diego CA: Evolutionary Programming Society, 1993.
2. Barr, Richard, et al. "Designing and Reporting on Computational Experiments with Heuristic Methods," *Journal of Heuristics* (1995). Kluwer Academic Publishers.
3. Brassard, Gilles and Paul Bratley. *Algorithms: Theory & Practice*. Englewood Cliffs, NJ: Prentice Hall, 1988.
4. Brinkman, Donald J. *Genetic Algorithms and Their Application to the Protein Folding Problem*. MS Thesis, AFIT/GCE/ENG/93D-02, EN, Wright Patterson Air Force Base, December 1993.
5. Brinkman, Donald J. and George H. Gates, Jr., "Implementation of a CHARMM Energy Function." In "/usr/genetic/Toolkit/CHARMM" directory on Thor, September 1994.
6. Brinkman, Donald J., et al. "Parallel Genetic Algorithms and Their Application to the Protein Folding Problem." *Proceedings of the Intel Supercomputer Users' Group 1993 Annual North America Users' Conference*, edited by JoAnne Wold. joanne@ssd.intel.com: Intel Supercomputer Systems Division, 1993.
7. Brooks, Bernard R., et al. "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations," *Journal of Computational Chemistry*, 4(2):187-217 (1983).
8. Brooks, Charles, et al. "Chemical Physics of Protein Folding," *Proceeding National Academy of Science*, 95:11037-11038 (September 1998).
9. C.A. Bohn, G.B. Lamont, J.K. Little and R. Raines. "Aysmmetric Load Balancing on a Heterogeneous Cluster of PCs," *Int'l Conference on Parallel and Distributing Processing Techniques and Applications*, 2515-2522 (1999).
10. Cahoon, J. P., et al. "A Multi-population Genetic Algorithm for Solving the K-Partition Problem on Hyper-cubes." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 244-248. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
11. Chan, Hue Sun and Ken A. Dill. "The Protein Folding Problem," *Physics Today*, 24-32 (February 1993).
12. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design*. Reading, MA: Addison-Wesley Publishing Company, August 1988.
13. CHARMM. *CHARMM User's Guide*.
14. Christine A. Orengo, annabel E. Todd and Janet M. Thornton. "From Protein Structure to Function," *Current Opinion in Structural Biology*, 9:374-382 (1999).

15. Committee on Physical, Mathematical, and Engineering Sciences. *Grand Challenges 1993: High Performance Computing and Communications*. Office of Science and Technology Policy, 1992.
16. Creighton, Thomas E. *Protein Folding*. New York: W.H. Freeman and Company, 1992.
17. Cresenzi, et. al. "On the Complexity of Protein Folding," ??, ??(??) (??).
18. David J. Brockwell, D. Alastair Smith and Sheena E. Radford. "Protein Folding Mechanisms: New Methods and Emerging Ideas," *Current Opinion in Structural Biology*, 10:16-25 (2000).
19. Davis, Lawrence. "Adapting Operator Probabilities in Genetic Algorithms." *International Conference on Genetic Algorithms*. 61-76. 1989.
20. Davis, Lawrence, editor. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold, 1991.
21. De Jong, Kenneth and William Spears. "On the State of Evolutionary Computation." *Proceedings of the Fifth Interantional Conference on Genetic Algorithms*, edited by Setphanie Forrest. 618-623. San Mateo, CA 94403: Morgan Kaufmann Publishers, Inc., July 1993.
22. De Jong, Kenneth A. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD dissertation, University of Michigan, 1975.
23. De Jong, Kenneth A. "Adaptive System Design: A Genetic Approach," *IEEE Transactions on Systems, Man and Cybernetics*, 10(9) (September 1980).
24. De Jong, Kenneth A. "On Using Genetic Algorithms to Search Program Spaces." *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. 210-216. Hillsdale, NJ 07642: Lawrence Erlbaum Associates, Publishers, 1987.
25. Deerman, Karl D. *Protein Structure Prediction Using Parallel Linkage Investigating Genetic Algorithms*. Masters, Computer Science, Air Force Institute of Technology, Wright Patterson Air Force Base, Dayton Ohio, March 1999.
26. Dorigo, Marco and Vittorio Maniezzo. "Parallel Genetic Algorithms: Introduction and Overview of Current Research," *Parallel Genetic Algorithms*, 5-35 (1993).
27. Duncan, Bruce S. "Parallel Evolutionary Programming." *The Second Annual Conference on Evolutionary Programming*. 202-208. San Diego, CA 92121: Evolutionary Programming Society, 1993.
28. Dymek, Andrew. *Examination of Hypercube Implementations of Genetic Algorithms*. MS thesis, -, EN, Wright Patterson Air Force Base, December 1992.
29. ECEPP/2. *ECEPP/2*.
30. Edwards, S. F. and P. W. Anderson. "Theory of spin glasses," *Journal of Physics*, 5:965-974 (1975).

31. El-Rewini, Hesham, et al. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall Series in Innovative Technology, Englewood Cliffs, NJ 07632: Prentice Hall, 1994.
32. Eshelman, Larry J., et al. "Biases in the Crossover Landscape." *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. David Schaffer. 10-19. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.
33. Eshelman, Larry J. and J. David Schaffer. "Preventing Premature Convergence in Genetic Algorithms by Preventing Incest." *Proceedings of the Fourth International Conference on Genetic Algorithms*. 115-122. San Mateo, CA: Morgan Kaufman Publishers, 1991.
34. Fogarty, Terence C. "Varying the Probability of Mutation in the Genetic Algorithm." *International Conference on Genetic Algorithms*. 104-109. 1989.
35. Fogel, David B. "Simulated Evolution: A 30-Year Perspective," *IEEE—ACSSC* (1990).
36. Fogel, David B. "On the Philosophical Differences between Evolutionary Algorithms and Genetic Algorithms." *The Second Annual Conference on Evolutionary Programming*. 23-29. San Diego CA: Evolutionary Programming Society, 1993.
37. Fogel, Lawrence J. "The Future of Evolutionary Programming," *IEEE—ACSSC*, 1036-1038 (1990).
38. Forrest, Stephanie and Melanie Mitchell. "Relative Building-Block Fitness and the Building-Block Hypothesis." *Foundations of Genetic Algorithms 2*. Morgan Kaufmann Publishers, Inc., 1993.
39. Garey, Michael R. and David S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. San Francisco, CA: W. H. Freeman and Company, 1979.
40. Gates, Jr., George H. "Combinatoric Algorithm (NP-Complete) Design Project, The Protein Folding Problem." CSCE 686 Advanced Algorithm Design, June 1994.
41. Gates, Jr., George H. *Predicting Protein Structure Using Parallel Genetic Algorithms*. MS Thesis, AFIT/GCS/ENG/94D-03, EN, Wright Patterson Air Force Base, December 1994.
42. Gates, Jr., George H., et al. "Parallel Simple and Fast Messy GAs for Protein Structure Prediction." *Proceedings of the Intel Supercomputer Users' Group 1995 Annual North America Users Conference*. Beaverton, Oregon: Intel Supercomputer Systems Division, 1995.
43. .Gaulke, Robert L. *The Application of Hybridized Genetic Algorithms to the Protein Folding Problem*. MS Thesis, AFIT/GCS/ENG/95D-03, EN, Wright Patterson Air Force Base, December 1995.
44. George H. Gates, Jr., et al. "Proceedings of the Second IEEE Conference on Evolutionary Computation." *Simple Genetic Algorithm Parameter Selection for Protein Structure Prediction*, edited by David Fogel. 1995.

45. Goldberg, David E. *Optimal Initial Population Size for Binary-coded Genetic Algorithms*. Technical Report TCGA Report Number 850001, University of Alabama, Alabama 35486: The Clearing House for Genetic Algorithms, Department of Engineering Mechanics, November 1985.
46. Goldberg, David E. *Genetic Algorithms in Search, Optimization & Machine Learning*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1989. Reprinted with corrections.
47. Goldberg, David E., et al. *Genetic Algorithms, Noise, and the Sizing of Populations*, chapter 6, 333-362. Complex Systems Publications, Inc., 1992.
48. Goldberg, David E., et al. "Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms." *Proceedings of the Fifth International Conference on Genetic Algorithms*, edited by Stephanie Forrest. 56-64. San Mateo, CA: Morgan Kaufmann Publishers, July 1993.
49. Goldberg, David E., et al. "Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale." *Complex Systems*. 415-444. 1990.
50. Goldberg, David E., et al. "Don't Worry, Be Messy." *International Conference on Genetic Algorithms*. 24-30. 1991.
51. Goldberg, David E., et al. "Messy Genetic Algorithms: Motivation, Analysis, and First Results." *Complex Systems*. 493-530. 1989.
52. Goldberg, David E. and Jon Richardson. "Genetic Algorithms with Sharing for Multimodal Function Optimization." *Proceedings of the Second International Conference on Genetic Algorithms*. 41-49. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1993.
53. Gordon, V. Scott and Darrell Whitley. "Serial and Parallel Genetic Algorithms as Function Optimizers." *Proceedings of the Fifth International Conference on Genetic Algorithms*, edited by Stephanie Forrest. 177-183. San Mateo, CA: Morgan Kaufmann Publishers, Inc., July 1993.
54. Grefenstette, J. J. "Optimization of Control Parameters for Genetic Algorithms," *IEEE Transactions on Systems, Man & Cybernetics*, 122-128 (1986).
55. Grefenstette, John J. "Learning by Analogy in Genetic Classifier Systems." *Proceedings of the Third International Conference on Genetic Algorithms*, edited by J. David Schaffer. 291-297. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.
56. Grefenstette, John J. *A User's Guide to Genesis 5.0*. Technical Report, Nashville, TN: Vanderbilt University, 1990.
57. Grefenstette, John J. "Lamarckian Learning in Multi-agent Environments." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 303-310. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
58. Grefenstette, John J. *Deception Considered Harmful*, 75-90. Foundations of Genetic Algorithms 2, Morgan Kaufmann, 1992.

59. Gustafson, John and Q. O. Snell, "HINT-A New Way to Measure Computer Performance." Proceedings of the HICSS-28 Conference in HTML, January 1995.
60. Harlan Crowder, Ron S. Dembo and John M. Mulvey. "On Reporting Computational Experiments with Mathematical Software," *ACM Transactions on Mathematical Software*, 5(2) (1979).
61. Hoare, C. A. R. *Communicating Sequential Processes*. London: Prentice-Hall International, 1984.
62. Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press, 1975.
63. Holland, John H. "Genetic Algorithms," *Scientific American*, 267(1):66-72 (July 1992).
64. Hornak, Joseph P., "The Basics of NMR." Internet, 1999. <http://www.cis.rit.edu/htbooks/nmr/inside.htm>.
65. Intel. *Paragon System Manual, I*. Intel Corp., 1988.
66. Intel. *iPSC/860 Basic Math Library User's Guide*, April 1991.
67. Jaenicke, R. "Protein Folding: Local Structures, Domains, Subunits, and Assemblies," *Biochemistry*, 30:3147-3161 (1991).
68. Jones, David T. "Protein Structure Prediction in the Postgenomic Era," *Current Opinions in Structural Biology*, 10:371-379 (2000).
69. Jr., Charles Kaiser. *Refined Genetic Algorithms for Polypeptide Structure Prediction*. Computer Science, AFIT/GCS/ENG/96D-13, School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson Air Force Base Ohio, December 1996.
70. Kaiser, Charles, et al. "Polypeptide Structure Prediction: Real-Value Versus Binary Hybrid Genetic Algorithms," *ACM Symposium on Applied Computing* (February 1997).
71. Kargupta, Hillol. *SEARCH, Polynomial Complexity, and The Fast Messy Genetic Algorithm..* Dissertation, University of Illinois at Urbana-Champaign, IL, USA, January 1996.
72. Kauffman, S. A. "Adaptation on rugged fitness landscapes," *Lectures in the Sciences of Complexity*, 1:527-618 (1989).
73. Kauffman, Stuart A. *The Origins of Order, Self-Organization and Selection in Evolution*. New York: Oxford Unbiversity Press, 93.
74. Kloeppel, James. "Fast Measurement Technique Reveals Early Steps in Protein Folding," *News from the University of Illinois at Urbana-Champaign* (December 1996).
75. Knjazew, Dimitri and David E. Goldberg. "OMEGA - Ordering Messy GA: Solving Permutation PROblems with the Fast Messy Genetic Algorithm and Random Keys," *GENETIC ALGORITHMS AND CLASSIFIER SYSTEMS*, 181-188

76. Kronsjö, Lydia and Dean Shumsheruddin, editors. *Advances in Parallel Algorithms*. New York: Halsted Press, 1992.
77. Kuck & Associates. *CLASSPACK Basic Math Library/C User's Guide* (Release 1.3 Edition). Champaign, IL 61820, November 1993.
78. Kumar, Vipin, et al. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.
79. Lamont, Gary B., et al. "Evolutionary Algorithms." Compendium of Parallel Programs for the Intel iPSC Computers, Volume V.
80. Larson, Roland E. and Robert P. Hostetler. *Calculus with Analytic Geometry* (3rd Edition). Lexington, MA: D.C. Heath and Company, 1986.
81. LeGrand, Scott M. and Kenneth M. Merz Jr. "The Application of the Genetic Algorithm to the Minimization of Potential Energy Functions," *Journal of Global Optimization*, 49-66 (1993).
82. Lengauer, Thomas. "Algorithmic Research Problems in Molecular Bioinformatics," *Arbeitspapiere der GMD 748* (May 1993).
83. Levi, Shem-Tov and Ashok K Agrwala. *Real Time System Design*. McGraw-Hill Computer Science Series, New York: McGraw-Hill Publishing Company, 1990.
84. Lewis, Ted G. and Hesham El-Rewini. *Introduction to Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall, 1992.
85. Loncharich, Richard J. and Bernard R. Brooks. "The Effects of Truncating Long-Range Forces on Protein Dynamics," *PROTEINS: Structure, Function, and Genetics*, 6:32-45 (1989).
86. Mascagni, Michael. "SPRNG: A Scalable Library for Pseudorandom Number," *SIAM* (1999).
87. Matsumoto, Makoto and Takuji Nishimura. "The Dynamic Creation of Distributed Random Number Generators," *SIAM* (1999).
88. Merkle, Laurence D. *Generalization and Parallelization of Messy Genetic Algorithms and Communication in Parallel Genetic Algorithms*. MS thesis, AFIT/GCE/ENG/92D-08, EN, Wright Patterson Air Force Base, December 1992.
89. Merkle, Laurence D. *Analysis of Linkage Friendly Genetic Algorithms*. PhD dissertation, Air Force Institute of Technology, 1996.
90. Merkle, Laurence D. and George H. Gates, Jr., "Paragon Implementation of a Parallel Fast Messy Genetic Algorithm." In `/usr/genetic/Toolkit/Messy/ParFast` directory on Thor, June 1994.
91. Merkle, Laurence D., et al. "Hybrid Genetic Algorithms for Minimization of a Polypeptide Specific Energy Model." *Proceedings of the Third IEEE Conference on Evolutionary Computation*. 1996.

92. Merkle, Laurence D., et al. "Hybrid Genetic Algorithms for Polypeptide Energy Minimization." *Applied Computing 1996: Proceedings of the 1996 Symposium on Applied Computing*. New York: The Association for Computing Machinery, 1996.
93. Merkle, Laurence D., et al. "Scalability of an MPI-based Fast Messy Genetic Algorithm," *ACM Symposium on Applied Computing* (March 1998).
94. Merkle, Laurence D. and Gary B. Lamont. "Comparison of Parallel Messy Genetic Algorithm Data Distribution Strategies." ??, ??, 1993. To be published.
95. Merkle, Laurence D. and Gary B. Lamont. "Comparison of Parallel Messy Genetic Algorithm Data Distribution Strategies." *Proceedings of the Fifth International Conference on Genetic Algorithms*, edited by Stephanie Forrest. 191-198. San Mateo, CA: Morgan Kaufmann Publishers, July 1993.
96. Merkle, Laurence D., Gates Jr., George H., Lamont, Gary B., and Pachter, Ruth. "Application of the Parallel Fast Messy Genetic Algorithm to the Protein Folding Problem." *Proceedings of the Intel Supercomputer Users Group 1994 Annual North America Users Conference*, edited by JoAnne Wold. 189-195. June 1994.
97. Merz, Jr., Kenneth M. and Scott M. LeGrand, editors. *The Protein Folding Problem and Tertiary Structure Prediction*. Boston: Birkhäuser, 1994.
98. Michael J.E. Sternberg, Paul A. Bates, Lawrence A. Kelley and Robert M. MacCallum. "Progress in Protein Structure Prediction: Assessment of CASP3," *Current Opinion in structural Biology*, 9:368-373 (1999).
99. Michalewicz, Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs* (2nd Edition). New York: Springer-Verlag, 1994.
100. Michaud, Steven R., et al. "A fmGA with Immunological EA," *GECCO*, ??(??):?? (March 2001).
101. Michaud, Steven R., et al. "Load Balancing Search Algorithms on a Heterogeneous Cluster of PCs," *Scientific IAM*, ??(??):?? (March 2001).
102. Mills, Richard L. *Statistics for Applied Economics and Business*. New York: McGraw-Hill Book Company, 1977.
103. Moul, John and Eugene Melamud. "From Fold to Function," *Current Opinion in Structural Biology*, 10:384-389 (2000).
104. Mühlenbein, H., et al. "The Parallel Genetic Algorithm as Function Optimizer," *Parallel Computing*, 17(7):619-632 (1991).
105. Nash, J. "Non-cooperative games," *Annals of Mathematics*, 54(2):286-295 (1951).
106. Nayeem, Akbar and others. "A comparative Study of the Simulated-Annealing and Monte Carlo-with-Minimization Approaches to the Minimum-Energy Structures of Polypeptides: [Met]-Enkephalin," *Journal of Computational Chemistry*, 12(5):594-605 (1991).

107. Neumaier, Arnold. "Molecular Modeling of Proteins and Mathematical Prediction of Protein Structure," *Society for Industrial and Applied Mathematics Review*, 39(3) (September 1997).
108. Ngo, Thomas J, et al. *Computational Complexity, Protein Structure Prediction, and the Levinthal Paradox*, chapter 14, 433–506. in [97], 1994.
109. Nielson, Gregory, et al. *Scientific Visualization Overviews, Methodologies, and Techniques..* Los Alamitos, CA.: IEEE Computer Society, 1997.
110. NPACI. "Parallel Random Number Generators," *Scientific Computing at NPACI (SCAN)* (1999). <http://www.npaci.edu/online/v3.7/SCAN1.html>.
111. Office of Technology Assessment. *Mapping Our Genes—The Genome Projects: How Big, How Fast?*. Technical Report No. OTA-BA-373, U. S. Government Printing Office, Washington, D.C.: U. S. Congress, 1988.
112. Olsan, James B. *Genetic Algorithms Applied to a Mission Routing Problem*. MS Thesis, AFIT/GCE/ENG/93-12, EN, Wright Patterson Air Force Base, December 1993.
113. Pacheco, P. *Parallel Programming with MPI* (1st Edition). ????: Morgan Kaufmann Publishers, 1996.
114. Pachter, Ruth, et al. "Smart Structures and Materials," *SPIE Proceedings 1* (1993).
115. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley Publishing Company, 1984.
116. Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
117. Pettey, Chrisila C. and Michael R. Leuze. "A Theoretical Investigation of a Parallel Genetic Algorithm." *Proceedings of the Third International Conference on Genetic Algorithms*. 398–405. San Mateo, CA: Morgan Kaufmann Publishers, Inc., June 1989.
118. Piccolboni, A. and G. Mauri. "Distance Space Evolutionary Algorithms for Protein Folding Predictions," *IEEE* (1998).
119. Ramachandran, et al. "Stereochemistry of Polypeptide Chain Configurations," *Journal of Molecular Biology*, 7:95–99 (1963).
120. Research, IBM. "Blue Gene Architecture," (August 2000).
121. Research, IBM. "IBM Unveils 100 Million Research Initiative to Build World's Fastest Supercomputer," (August 2000).
122. Research, IBM. "Protein Folding," (August 2000).
123. Richard H.F. Jackson, Paul T. Boggs, Stephen G. Nash and Susan Powell. "Guidelines for Reporting Results of Computational Experiments. Report of the Ad Hoc Committee," *Mathematical Programming*, 49:413–425 (1991).

124. Rupp, Bernhard, "X-ray Crystallography 101." Internet, Mar 1999. <http://www-structure.llnl.gov/Xray/101index.html>.
125. Russell, Robert B. and Chris P. Ponting. "Protein Fold Irregularities that Hinder Sequence Analysis," *Current Opinion in Structural Biology*, 8:364-371 (1998).
126. Rutchik, Robert H. and Jay Casselberry, "EIA Guidelines for Statistical Graphs." <http://www.eia.doe.gov/neic/graphs/preface.htm>, May 1999.
127. S. Michaud, J. Zydallis, G. Lamont and R. Pachter. "Detecting Secondary Peptide Structures by Scaling a Genetic Algorithm," *PP01* (2001).
128. Sarah a. Teichmann, Cyrus Chothia and Mark Gerstein. "Advances in Structural Genomics," *Current Opinion in structural Biology*, 9:390-399 (1999).
129. Sawyer, George A., et al., "Hypercube Implementation of a Parallel Simple Genetic Algorithm." In *"/usr/genetic/Toolkit/PSGA"* directory on Thor, April 1994.
130. Schaffer, J. David, et al. "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization." *Third International Conference on Genetic Algorithms*. 51-60. 1989.
131. Schaffer, J. David and Larry J. Eshelman. "On Crossover as an Evolutionarily Viable Strategy." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 61-68. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
132. Schulze-Kremer, Steffen. *BioComputing Hypertext CourseBook, Chapter 5*. www.techfak.uni-bielefeld.de/bcd/Curric/ProtEn/protein.html: ???, ???
133. Spears, William M. and Kenneth A. De Jong. "Using Genetic Algorithms for Supervised Concept Learning," *IEEE-CH*, 29(15):335-341 (July 1990).
134. Spiessens, Piet and Bernard Manderick. "A Massively Parallel Genetic Algorithm: Implementation and First Results." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 279-285. San Mateo, CA: Morgan Kaufmann Publishers, 1991.
135. Srinivas, M. and Lalit M. Patnaik. "Genetic Algorithms: A Survey," *COMPUTER*, 27(6):17-26 (June 1994).
136. Starkweather, T., et al. "A Comparison of Genetic Sequencing Operators." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 69-76. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
137. Sterling, Thomas, et al. *Enabling Technologies for Peta(FL)OPS Computing*. Technical Report CCSF-45, California Institute of Technology, July 1994.
138. Stigers, Kimberly D., et al. "Designed Molecules that Fold to Mimic Protein Secondary Structures," *Current Opinion in Chemical Biology*, 3:714-723 (1999).
139. Stryer, Lubert. *Biochemistry* (3rd Edition). New York: W.H. Freeman, 88.

140. Stryer, Lubert. *Biochemistry* (4rd Edition). New York: W.H. Freeman, 95.
141. Tanese, Reiko. "Parallel Genetic Algorithms for a Hypercube." *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, edited by John J. Grefenstette. 177-183. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, July 1987.
142. Toran, Jacobo. *P-Completeness, Chapter 9*. ??, ??
143. Tufte, Edward. "Presenting Data and Information." January 2000.
144. Whitley, Darrel. "The *GENITOR* Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best." *International Conference on Genetic Algorithms*. 1989.
145. Whitley, Darrell, et al. "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator." *Proceedings of the Fourth International Conference on Genetic Algorithms*, edited by Richard K. Belew and Lashon B. Booker. 133-140. San Mateo, CA: Morgan Kaufmann Publishers, July 1991.
146. Wilson, Gregory, "The History of the Development of Parallel Computing." <http://parallel.ru/history/wilson/history.html>, November 1994.
147. Wolpert, David H. and William G. Macready. "No Free Lunch Theorems for Search,"

Vita

Captain Steven Ronald Michaud enlisted in the US Air National Guards in 1980 and entered active duty in 1981. During the next twelve years he served as a Telecommunications Information Specialist, Technical Controller, and Systems Programmer, rising in rank from Airman Basic to Technical Sergeant. While on active duty he received an Bachelor Degree in Computer Science, summa cum laude from Park College, Missouri, in November 1993. He was selected for Officer Training School (OTS) the following month and finished OTS in July of 1994 at Medina Air Station, San Antonio Texas.

Upon completion of OTS, Steven was assigned to support the Secretary of Defense computer support systems. He later finished a Master's Degree in Computer Resource Management from the Webster University, Oklahoma in September 1996. In mid-1997, Steven was reassigned as the executive officer for the 425th Air Base Squadron for one year, whereupon he was made the Chief, Communications Section responsible for all communication support in Izmir, Turkey. He then was reassigned to the Air Force Institute of Technology (AFIT), but before attending AFIT he went to Squadron Officer School in route and aided his flight in receiving the highest flight award, the Chief of Staff Award, which is awarded to the best overall flight in the class.

Steven's military assignment's include Langley AFB, Va., Hill AFB, Ut, Sixth Allied Tactical Air Force, Turkey, Tinker AFB, Ok., Ankara AS, Turkey, Air Force Pentagon Communications Agency, Va., 425th ABS, Turkey. His follow-on assignment is the Maxwell AFB where he will serve in CADRE's Wargaming Center.

Permanent address: Wright Patterson Air Force Base, Dayton Ohio, 45433

REPORT DOCUMENTATION PAGEForm Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) 20-03-2001		2. REPORT TYPE MASTER'S THESIS		3. DATES COVERED APR 00 - MAR 01	
4. TITLE AND SUBTITLE SOLVING THE PROTEIN STRUCTURE PREDICTION PROBLEM WITH FAST MESSY GENETIC ALGORITHMS (SCALING THE FAST MESSY GENETIC ALGORITHM TO MEDIUM-SIZED PEPTIDES BY DETECTING SECONDARY STRUCTURES)				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
				5d. PROJECT NUMBER	
6. AUTHOR(S) STEVEN R. MICHAUD, CAPT, USAF				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P. Street, Bldg 640 Wright-Patterson AFB, OH 45433-7756				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-06	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Lab, Materials Division AFRL/MLPJE Attn: Dr. Ruth Pachter, ruth.pachter@wpafb.af.mil 3005 P Street B651 R189 Wright-Patterson AFB, OH 45433 DSN: 785-3808 x3177				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION UNLIMITED					
13. SUPPLEMENTARY NOTES Advisor: Dr. Gary B. Lamont, DSN 785-3636 x4718, gary.lamont@afit.af.mil					
14. ABSTRACT The ability to accurately predict a polypeptide's molecular structure given its amino acid sequence is important to numerous scientific, medical, and engineering applications. Studies have been conducted in the application of Genetic Algorithms (GAs) to this problem with promising initial results. In this thesis report, we use the fast messy Genetic Algorithm (fmGA) to attempt to find the minimization of an empirical CHARMM energy model and generation of the associated conformation. Previous work has shown that the fmGA provided favorable results, at least when applied to the pentapeptide [Met]-Enkephalin. We extend these results to a larger Polyalanine peptide by utilizing secondary structure information as both searching constraints and seeding the initial population. Additional efforts were conducted to improve the performance of the algorithm with respect to solving the Protein Structure Prediction (PSP) problem through a short-circuiting operator--where complete evaluation of the fitness function is halted if initial results are not promising, and by conducting additional searches on faster machines in a heterogeneous environment. Results indicate that, on average, this localized search tends to produce better final solutions. Finally, the fmGA as applied to the PSP problem is analyzed and shown to have improved performance and effectiveness.					
15. SUBJECT TERMS Gentic Algorithms, Evolutionary Algorithms, Protein Structure Prediction, Secondary Structures, Met-Enkephalin, Stochastic Search Methods					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 259	19a. NAME OF RESPONSIBLE PERSON Dr. Gary B. Lamont, ENG
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 785-3636 x4718